2

Naval Research Laboratory

Washington, DC 20375-5000

NRL Memorandum Report 6820

User's Guide for SAMUEL, Version 1.3

JOHN J. GREFENSTETTE AND HELEN G. COBB

Navy Center for Applied Research in Artificial Intelligence
Information Technology Division

May 6, 1991

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 1991 May 6	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE User's Guide for SAMUEL, Version 1.3	5. FUNDING NUMBERS PE - 62234N TA - RS34-C74-000 JO. # - 55-0230-0-1
6. AUTHOR(S) John J. Grefenstette and Helen G. Cobb	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5000 ATTEN: Code 5510	8. PERFORMING ORGANIZATION REPORT NUMBER NRL Memorandum Report 6820
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. *SUPPLEMENTARY NOTES*	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

SAMUEL is a machine learning system designed to actively explore alternative behaviors in a simulated environment, and to construct high performance rules from this experience. The learning method relies on the notion of competition and employs genetic algorithms to search the space of decision policies. The rule language in SAMUEL also makes it easier to incorporate existing knowledge, whether acquired from experts or by symbolic learning programs. The system includes a competition based production system interpreter, incremental strength updating procedures to measure the utility of rules, and genetic algorithms to modify strategies based on past performance. The current version includes a more convenient language for the expression of tactical control rules, better interfaces, and a number of new heuristics for rule modification. We have experimented with SAMUEL on a task involving learning control rules that enable a simulated robotic aircraft to evade an approaching missile. SAMUEL has been able to learn high performance strategies for this task. This manual should help the user to experiment with SAMUEL on other problems.

14. SUBJECT TERMS Machine learning Genetic algorithms Reinforcement learning			15. NUMBER OF PAGES 96 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

NSN 7540-01-280-5500 Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

CONTENTS

USER'S GUIDE FOR SAMUEL, VERSION 1.3

CHAPTER 1

INTRODUCTION

This document is the User's Manual for the SAMUEL system. SAMUEL stands for Strategy Acquisition Method Using Empirical Learning.[1]

PURPOSE OF THE SAMUEL SYSTEM

SAMUEL learns condition-action rules from a computer simulation of a problem domain. For example, the problem specific module accompanying this distribution models the Evasive Maneuvers (EM) problem. In this 2-dimensional computer simulation of air combat, SAMUEL learns rules for controlling the turning rate of a plane so the plane avoids being hit by an approaching missile. SAMUEL can learn the condition-action rules for a variety of problems. In general, SAMUEL can learn sequential decision rules for any problem domain where the overall evaluation of the decisions occurs at the end of the sequence.

IMPLEMENTATION LANGUAGE OF SAMUEL

SAMUEL is written in the C programming language. The makefile accompanying the distribution has compilation flags for the Sun3, the Sun4 SPARCSTATION, and the Butterfly's Mach Uniform System. Users of other systems may need to tailor the C code to suit their own environments.

OBJECTIVES OF THE MANUAL

This manual is intended primarily for researchers and application developers. By examining the application of SAMUEL to the EM problem domain, we illustrate how a user might apply SAMUEL to other problems. The manual does not explicitly address how to make modifications to SAMUEL that extend beyond the problem domain modules. However, the manual does provide an overall description of the system that would be useful background reading to users who intend to make substantial modifications to SAMUEL. For additional information on SAMUEL, the user should consult the Reference section of this manual.

[1]The name also honors Art Samuel, one of the pioneers in machine learning.

Manuscript approved February 8, 1991

USE OF SAMUEL

The SAMUEL system and its accompanying documentation are the property of the Department of the Navy. This system is to be used for official Department of Navy purposes only. The use of the SAMUEL system is governed by a formal Memorandum of Agreement between the Navy Center for Applied Research in Artificial Intelligence (NCARAI) and the user's transition site. Researchers and application developers who make use of SAMUEL in their research products or application products should credit the program author, John J. Grefenstette, and the Naval Research Laboratory.

The author of SAMUEL and the authors of its accompanying documentation do not guarantee the program code or the SAMUEL User's Guide to be error free or appropriate for any particular purpose. To report any problems, see the Program Maintenance and Support section of this manual.

ORGANIZATION OF THE MANUAL

The rest of this manual is organized as follows. Chapter 2 provides a user with a quick overview of the system. Chapters 3 through 6 expand on Chapter 2, giving a review of the system's knowledge structures and architecture. In addition to discussing the Problem Specific Module in general, Chapter 6 also outlines the essential procedures, functions, and variables for developing a world model (of a problem domain) to interface with SAMUEL's Performance Module. Chapter 7 introduces a user in a hands-on fashion to SAMUEL. This chapter includes a demonstration of SAMUEL's display output and a simple example experiment using the EM environment. Chapter 8 gives program support information. The appendices summarize more detailed information. Appendices A through C describe information pertaining to input files used by SAMUEL: Appendix A lists SAMUEL's input parameters; Appendix B describes the syntax used for defining sensors and controls in the world model, and Appendix C illustrates the syntax of rules that can optionally be read into SAMUEL during program initialization. Appendix D gives implementation details of SAMUEL, listing by file the procedures, functions, and variables of the system.

A user should read the chapters of this manual in their order of presentation. However, reading about how to interface a user's domain model to SAMUEL is not essential to gaining familiarity with the system (Chapter 7).

This manual uses certain conventions of presentation. Key concepts presented in the manual are placed initially in italics for emphasis. Operator names used in the Learning Module are always in italics. Actual parameters, procedures, functions, variables, and file names are in bold type.

CHAPTER 2

SYSTEM OVERVIEW

Figure 1 shows the architecture of the SAMUEL system. SAMUEL consists of a *Performance Module*, a *Learning Module*, and a *Problem Specific Module*.

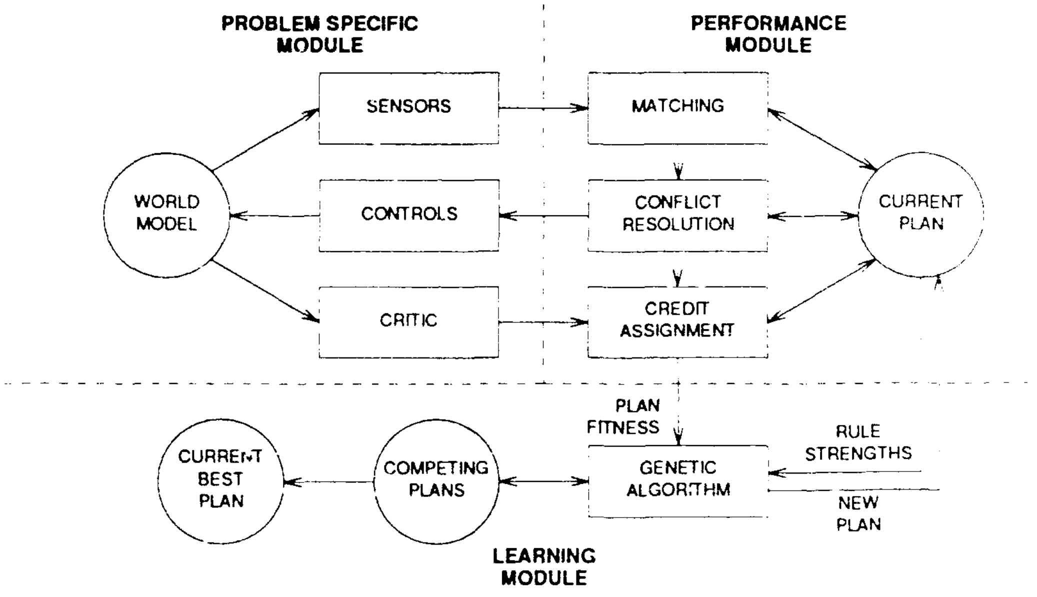


Figure 1. Architecture of the SAMUEL System

KNOWLEDGE REPRESENTATION

SAMUEL uses three primary knowledge structures within these modules: a *population*, *plans*, and *rules*. The Learning Module examines population of plans that changes over time; the population consists of a set of plans; each plan consists of a set of rules. Each generation of the genetic algorithm, the Learning Module searches for the best performing members among a *population* of plans. To accomplish the search, the Performance Module, in conjunction with the Problem Specific Module, evaluates the performance of each *plan*. Plans describe strategies for interacting with the world in terms of condition-action rules. Depending on the world's state, the Performance Module selects a *rule* from the current plan that specifies the control actions to be applied in the world model of the Problem Specific Module.

PERFORMANCE MODULE

The Performance Module is a Competition-based Production System (CPS) that interacts with both the world model through the Problem Specific Module interfaces and the Genetic Algorithm (GA) of the Learning Module. The CPS module has three major functions: *matching*, *conflict resolution*, and *credit assignment*. The CPS's *matching* and *conflict resolution* functions essentially perform the match/resolve-conflicts/act cycle of the traditional production system cycle. The Problem Specific Module carries out the action selected by the CPS. During matching, the CPS examines each rule of a plan and determines the degree of match between the conditions of a rule and the currently sensed world values. During conflict resolution, the CPS first selects an action-value among those recommended by a set of highly matching rules based on rule *strengths* and then communicates this value to the Problem Specific Module. If there is more than one control action, the CPS selects, in a similar manner, a value for each of the other control actions. The production cycle repeats until the completion of an *episode* in the Problem Specific Module. The CPS next executes the *credit assignment* function, adjusting the relative strengths of rules in the current plan based on rule performance of during the episode. The CPS repeats the execution of its three major functions for a number of episodes to obtain an *average payoff*. Finally, the CPS communicates the average payoff of the current plan to the Learning Module.

PROBLEM SPECIFIC MODULE

The Problem Specific Module consists of a *world model* simulation and the *critic*, *sensor*, and *control* interfaces. The *sensor interface* communicates the world's current conditions to the Performance Module's *matching* function. In turn, the Performance Module's *conflict resolution* function communicates the currently selected values of the actions to the *control interface*. Given the environmental state and the selected values of the control actions, the *world model* computes the next time step's environmental state. Under conditions determined by the world model, the episode ends. As a result, the *critic* communicates its evaluation of the plan, called *payoff*, to the Performance Module. Between episodes, the Problem Specific Module reinitializes the world state to one of the possible initial conditions described in the problem definition.

LEARNING MODULE

The Learning Module, based on the standard GA of the GENESIS package (Grefenstette, 1983), incorporates additional learning heuristics over the standard GA. The GA develops high performance reactive strategies called plans through the competition of plans within a population. Every generation, the Learning Module sends each of its *competing plans* to the Performance Module for evaluation. The Performance Module returns the average payoff of each plan. The GA determines the *fitness* of a plan by scaling this average payoff to a *baseline performance*. Based on the fitness evaluations of the competing plans, the GA then chooses relatively high performing plans for reproduction using the *select* operator. The GA applies genetic operators such as *crossover* and *mutation* to copies of these selected plans to produce plausible new plans for the next generation. The GA's generational cycle repeats until one of the user-specified stopping criteria is satisfied. Depending on options specified by a user, the Learning Module might periodically save the *current best plan* and other statistics.

EVALUATION LOOPS IN SAMUEL

There are four nested evaluation loops in SAMUEL: *experiments*, *generations*, *trials*, and *episodes*. A user typically evaluates SAMUEL for a number of repetitions called *experiments* in order to obtain a statistical estimate of SAMUEL's performance in solving a problem. The evaluation of all members in the GA's population corresponds to a *generation*. Each generation, the GA evaluates a population in order to breed potentially better performing plans for the next generation. The evaluation of each population member corresponds to a *trial*. Each trial, the CPS evaluates a plan by measuring the plan's average performance over a number of episodes. (The trial counter continues without reinitialization after each generation.) An *episode* is the decision sequence for accomplishing a domain task. During an *episode*, the CPS also evaluates the rules of a plan by distributing the payoff received from the world model's critic function over the *active rules* used in the decision process. (Each episode lasts for a number of decision *steps*. However, there is no evaluation associated with a single decision step.)

LEVELS OF LEARNING IN SAMUEL

Learning in SAMUEL occurs at two distinct levels: *credit assignment* at the rule level, and *genetic competition* at the plan level.

The CPS, through credit assignment, adjusts the relative strengths of rules within a plan using the payoff of an episode. The strength of a rule serves as a prediction of the expected level of payoff that would be achieved when the rule fires. Thus, the CPS learns rules that are more likely to result in episodes yielding high payoff.

These payoff values also provide the basis for evaluating a plan's performance in the Learning Module. The GA computes each plan's fitness based on its average payoff. This fitness value provides an estimate of the plan's relative competitiveness with respect to other plans in the population. As a result, SAMUEL learns plans that yield overall high performances as well as the rules in those plans that yield high payoff.

CHAPTER 3

KNOWLEDGE REPRESENTATION

As stated in the Introduction, there are three knowledge structures in SAMUEL: the population, the plans making up the population, and the rules defining each plan.

POPULATION

Throughout a program run, the population consists of a fixed number of plans. All plans within the population are initially the same.

PLAN

A plan consists of three parts: (1) a *rule set*, (2) the *associated properties* describing the plan and its history, and (3) a *vector of operator probabilities*. The GA uses these associated properties and the vector of operator probabilities in its generational cycle.

The Rule Set

A plan has a variable number of rules. Initially, a plan may have one or many rules depending on the initialization options the user chooses. During program execution, the GA operators of the Learning Module generate new rules for plans based on the plan's experience in interacting with the Problem Specific Module. (If the *delete* operator is active, the GA may occasionally remove a rule from a plan.)

Associated Properties

value

The CPS evaluates a plan by finding the average payoff of the plan over a number of episodes. The *value* property stores this average payoff.

fitness

The GA computes the *fitness* property of a plan using the plan's value property and the baseline performance of the current generation. The fitness property stores the plan's fitness value.

offspring

The *offspring* property saves the expected number of offspring the plan can expect to create. This expected number is based on the average fitness of the population's plans.

length

The *length* property indicates the current number of rules in the plan.

gen

The *gen* property indicates the generation the plan was created.

trial

The *trial* property indicates the trial the plan was created.

parent1 and parent2

The *parent1* and *parent2* properties record the identification numbers of the parent plans

Vector of Operator Probabilities

The operator probability vector holds the probability of applying each of the GA operators. Table 1 lists the order of the operators in the vector.

Index	Operator Probability
0	No operator.
1	*mutate*
2	*crossover*
3	*specialize*
4	*generalize*
5	*creep*
6	*delete*
7	*merge*

Table 1. Vector Operator Probabilities

The *mutate* probability is the probability of mutating a plan; there is a separate probability for mutating a rule.

RULES

Each rule has two parts: (1) a *condition-action production*, and (2) the *associated properties* of the rule. For example, Rule 10 might be

```
Rule 10
    if  (and (range 250 1000) (speed 500 1200))
    then (and (turn 0 90))
    parent 0   created 0   fixed 1   matched 0   partially-matched 0
    action 1   bid 0   active 0   fired 0 mean 500   var 0   strength 500   act 0.5   bias 0
```

The associated properties list records the rule's history, and for each control action, the statistics that reflect the rule's relative competitiveness. The CPS uses these associated properties in its production cycle.

Condition-Action Production

Each condition-action production has the form

if (and $c_1 \cdots c_n$)
then (and $a_1 \cdots a_m$)

where each c_i is a condition, and each a_j is an action. For example, a condition-action production might be

```
if  (and (range 250 1000) (speed 500 1200))
then (and (turn 0 90))
```

Conditions and actions are defined in terms of *attributes*. There are two kinds of attributes: *sensors* and *controls*. Each c_i in a rule specifies a condition on one of the sensors; each a_j specifies the settings for one of the controls. For example, the "range" and "speed" conditions in the rule above are sensor attributes. The "turn" action is a control attribute. An attribute has three components:

1. *name*,
2. *type*, and
3 *range of values*.

Conditions and actions together comprise the atomic formulas, or *atoms*, of the production rules in SAMUEL. For example,

(range 250 1000)

is an atom in the example above. The syntax of an atom depends on the attribute's type.

Name Component

The names of the sensors and controls describe their functions in the world model. A user must define control and sensor variables as part of specifying the world model. In the example above, the name of the first sensor attribute is "range."

Types of Attributes and Their Range of Values

SAMUEL supports four types of attributes: *linear*, *cyclic*, *structured*, and *pattern* Both sensor and control attributes can have types linear, cyclic, and structured. Currently, the pattern type only applies to sensor attributes. A user defines the attributes associated with each sensor and control variable in the **attributes** file. Appendix B describes the **attributes** file in more detail. A brief description of each kind of attribute follows.

Linear

Linear attributes take on linearly ordered numeric values from a fixed range of values. In addition to the name, the atom of a linear attribute specifies the attribute's upper and lower bounds. The user can divide the range up to a maximum of 255 segments. The number of segments determines the resolution of the sensor or control. The endpoints of each segment constitute the legal bounds in the atoms.

For example, the "range" condition in the rule above has a *type* = linear. Suppose the low and high values for "range" are 0 and 1000, respectively, and the number of segments partitioning "range" is 20. Another possible atom for "range" might be

(range 100 250)

A match of the "range" condition occurs when $100 \leq range \leq 250$.

The "turn" action in the rule above also has an attribute of *type* = linear. Should the *range* and *speed* sensors match the conditions in the rule's "range" and "speed" atoms, then the turn control would take on some value between 0 and 90.

Cyclic

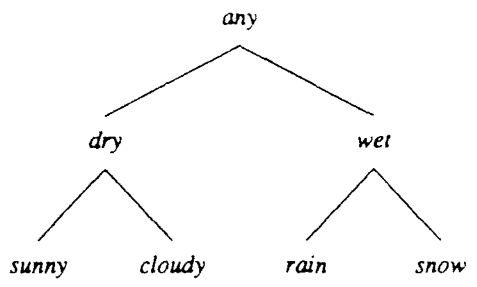
Cyclic attributes take on cyclically ordered numeric values. Like linear attributes, cyclic attributes have equally divided segments whose endpoints constitute the legal bounds in the atoms. Unlike linear attributes, any pair of legal values form the bounds of a valid cyclic atom. For example, suppose a cyclic *heading* sensor describes a vehicle's direction. If the atom is

(heading 330 90)

then a match of this "heading" condition occurs when the *heading* sensor has a value in either of two ranges: $330 \leq heading \leq 360$ or $0 \leq heading \leq 90$.

Structured

Nominal attributes take on values from the nodes of a tree-structured hierarchy. The atom of structured attribute contains a list of these nodes. For example, a sensor called *weather* may take on values from the following hierarchy:



An atom for the "weather" condition might be

(weather is [cloudy wet])

This list implies all the subnodes of the nodes listed. A match occurs in this example when the *weather* sensor is "cloudy", "wet", "rain", or "snow." A structured attribute can have at most 64 nodes in its hierarchy. The label of a node can have a maximum string length of 16 characters.

Pattern

Pattern attributes take on string values, or *patterns*, from the alphabet {0, 1, #} as in classifier systems (Holland, 1986). For example, the sensor *detector1* might be defined as an eight bit string. A possible atom for the "detector1" condition in a rule might be

(detector1 00##10#1)

A match occurs when all positions of the *detector1* sensor match the positions occupied by 0 and 1 in the atom's pattern.

Associated Properties

There are many associated properties of a rule. *Parent, created,* and *fixed* are properties pertaining to the GA that describe the rule's history. *Matched* and *partially-matched* are properties pertaining to the CPS that indicate how often the rule's conditions match sensor values.

The remaining properties, except for *bias*, pertain to the CPS's Profit Sharing Plan (PSP). The *bid, active* and *fired* properties are count statistics reflecting the degree of a rule's participation in the rule competition. The *mean, var,* and *strength* properties reflect the payoff associated with the rule's use. The *act* property indicates the level of a rule's use over time.

The *bias* property pertains to the CPS's Bucket Brigade Algorithm (BBA). The BBA is not included in this version of SAMUEL. A user should ignore the *bias* property.

parent

The *parent* property stores the identification number of the predecessor rule. For example, if SAMUEL creates Rule 12 by mutating Rule 10, then *parent* = 10.

created

The *created* property is the concatenation of a rule's generation time with a code indicating how SAMUEL created the rule. For example, if *created* = 321, then SAMUEL created the rule during generation 32 using the *mutation* operator (code 1). Codes for the operators are as follows: (1) *mutation*, (2) *crossover*, (3) *specialize*, (4) *generalize*, (5) *creep*, (6) *delete*, and (7) *merge*. See Chapter 5 for a discussion of the GA's learning operators.

fixed

The *fixed* property is a flag indicating that SAMUEL will not subject the rule to the GA operators that form new replacement rules by modifying existing rules (i.e., the *mutation* and *creep* operators). The fixed property provides a mechanism for inserting default rules into SAMUEL that remain unchanged by the GA. A user may initialize a rule to be fixed during program input. See the Appendix A for more details on the **params** file.

matched

The *matched* property is the number of times since the rule's creation that the rule's conditions have completely matched sensor readings. Completely matching rules are always included in the *match set*. See Chapter 4 for a discussion of matching and match sets.

partially-matched

Rules that partially-match have conditions that match some, but not all, of the current sensor readings. If there are no completely matching rules, then the *match set* consists of the best partially-matching rules. The *partially-matched* property indicates the number of times the rule has been a member of the match set as a partially-matched rule.

bid

The *match set* may have rules indicating several competing values for a control action. Each of these values bids to be selected. The *bid* property indicates the number of times that the value of a control action bids to be selected since its creation. See Chapter 4 for a discussion of action bidding.

active

If the value of a control action wins a bid, then all rules in the *match set* having that value for the control action are *active*. The *active* property indicates the number of times the rule has been active since its creation. See Chapter 4 discussion of rule firing.

fired

The *fired* property indicates the number of times the rule has won the bidding process since its creation.

mean

The *mean* property saves the time-averaged mean of the rule strengths. See Chapter 4 for a discussion on updating rule strengths.

var

The *var* property saves the time-averaged variance of the rule strengths.

strength

The *strength* property serves as a prediction of the rule's utility (Grefenstette, 1988). Thus, rule strengths are used in bidding process for rule firing. Each control action may be represented by several active rules in the match set. The control action bids the strength of the highest strength rule having that control action in the match set. See Chapter 4 for a discussion of updating rule strengths.

act

The *act* property indicates a rule's recent firing activity level using a metric ranging from 0 to 1. Rules initially have an activity level of 0.5. Each generation, SAMUEL decays the activity level of all rules by 0.95. When a rule fires, SAMUEL increases the rule's activity level by setting the activity to 0.1 plus 90 percent of the rule's current activity level. As a result, activity near 1 indicates recent firing of the rule, whereas activity near 0 indicates recent inactivity of the rule. Rules with low activity are subject to deletion if the *delete* operator is functioning.

bias

The *bias* property indicates the relative competitiveness of a rule in the Bucket Brigade Algorithm. The *bias* property may be used in future versions of SAMUEL.

CHAPTER 4

THE PERFORMANCE MODULE: THE PRODUCTION SYSTEM CYCLE

CONDITION MATCHING

Several rules may have conditions that *match* or *partially match* the sensor readings. Partial matching occurs when only some conditions match sensor values. The number of matching conditions in a rule is the *match score.* The largest possible match score is equal to the number of sensors. The CPS places all rules having the highest match score into a *match set.* For example, suppose *range* = 500 and *speed* = 450. Consider the following three hypothetical rules:

```
Rule 62
          if   (and (range 700 1500) (speed 150 700))
          then (and (turn -45 45))
          parent 50   created 0   fixed 1   matched 440   partially-matched 440
          action 1   bid 497   active 457   fired 289   mean 982   var 13300
                strength 866   act 0.5   bias 901

Rule 77
          if   (and (range 200 1000) (speed 50 550))
          then (and (turn 45 90))
          parent 10   created 0   fixed 1   matched 3468   partially-matched 253
          action 1   bid 3197   active 1766   fired 1506   mean 994   var 2622
                strength 943   act 0.5   bias 952

Rule 100
          if   (and (range 450 1200) (speed 400 1000))
          then (and (turn 90 135))
          parent 83   created 0   fixed 1   matched 284   partially-matched 338
          action 1   bid 132   active 375   fired 83   mean 971   var 21604
               strength 824   act 0.5   bias 904
```

In Rule 62, the "range" condition does not match on the *range* sensor. In this example, Rules 77 and 100 form the match set since both the "range" and "speed" conditions match the *range* and *speed* sensors. Suppose, instead, that these three rules had conditions that matched on only one sensor, i.e., *partially matched.* Then all of the rules would be placed in the match set. In general, if some rule matchs m out of n sensors, and no rule matches $m + 1$ sensors, then the match set consists of all rules with match score m.

Associated with each rule in the match set are the values of each control (i.e., action-values). In the example above, there is only one control action: *turn.* If the *turn* control is of type linear with a range from -180 to 180 quantized into segments of 45 degrees, then the action-values for the *turn* control for Rule 77 are 45 and 90. Similarly, the action-values for Rule 100 are 90 and 135.

CONFLICT RESOLUTION: ACTION BIDDING AND RULE FIRING

The *conflict resolution* of choosing a rule to fire is based on rule strengths. The action-values for each control *bid* to be selected. An action-value's bid is the strength of the *strongest rule* in the match set having that action-value. For example, assume Rules 77 and 100 above form the match set. In this case, action-value 90 would bid 943, the strength the stronger rule, Rule 77. However, since action-value 135 only occurs in one rule of the match set (Rule 100), the bid of action-value 135 is only 824.

Using Monte Carlo techniques, the CPS selects a winning action-value from a probability distribution formed using these bids. This procedure is unlike classifier systems in which all members of the match set vote on the action to be performed (Riolo, 1988). Using a distribution based on strengths of only the strongest rules prevents a large number number of low strength rules from combining to suggest an action-value that is actually associated with low payoff. All rules in the match set that have an action-value agreeing with the winner of the bid are said to be *active* (Wilson, 1987). The active rule winning the bid is said to *fire.*

If there is more than one action (i.e., control), SAMUEL assumes each action to be independent of the others. In other words, SAMUEL repeats the bidding process for each action. For example, suppose there are two actions. The three hypothetical rules above would have an additional line for "action 2." It is possible for one rule in the match set to fire the for first action, and another rule in the match set to fire for the second action. Each of these actions would have separate strengths associated with them.

CREDIT ASSIGNMENT: UPDATING RULE STRENGTH

The production cycle of match/resolve-conflicts/act repeats until the world model indicates the end of an episode. At that time, the *critic* of the Problem Specific Module returns a *payoff* value for the episode. The CPS's Profit Sharing Plan (PSP) performs the *credit assignment* function. Using the PSP, the CPS incrementally adjusts the strength of all active rules to reflect the current payoff, since any active rule could have potentially fired. The general approach for adjusting rule strength is to first reduce the rule's current strength by some fraction, and then increase the rule's strength by the same fraction of the payoff. Suppose R_1 -> R_2 -> R_3 is one of the active rule sequences considered during an episode having three steps. Also, suppose the fraction, c, is 0.10, and the payoff, r, is 100. Figure 2 illustrates the original rule strengths at time t and the updated rule strengths at the end of the episode (at $t + 3$).

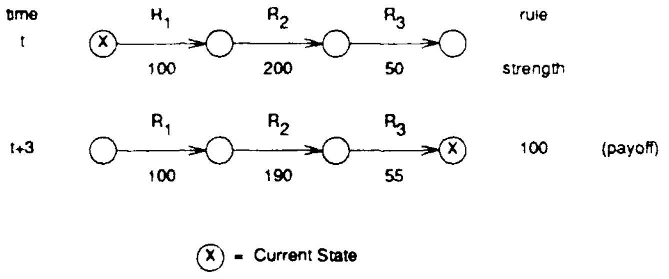


Figure 2. Updating the Strength of Rules

Notice that the rule correctly predicting the payoff, R_1, retains its original strength; the rule that overestimates the payoff, R_2, loses its strength, and the rule that underestimates the payoff, R_3 gains strength. Over the course of many episodes, the strengths of active rules converge to the expected level of payoff (Grefenstette, 1986). This observation motivates the use of rule strength during conflict resolution.

Instead of considering only the current adjustment of the rule strengths, the CPS uses a time-weighted averages of these adjustments. Specifically, the CPS uses a time-weighted average to estimate both the mean, μ_i and the variance, σ_i^2, of the payoff associated with each rule, R_i. Given an end-of-episode payoff, r, a fraction, c (c is the runtime parameter called the **psprate**), then μ_i is

$$\mu_i = (1 - c)\mu_i + cr.$$

Likewise, σ_i^2 is

$$\sigma_i^2 = (1 - c)\sigma_i^2 + c(\mu_i - r)^2.$$

Given μ_i and σ_i^2, the estimated strength of R_i is

$$strength\,(R_i) = \mu_i - \sigma_i.$$

The *strength function* ensures that a high strength rule must have both a high mean and a low variance. The CPS records the μ_i, σ_i, and the result of the *strength function* in the rule's associated properties *mean*, *var*, and *strength*, respectively.

CHAPTER 5

THE LEARNING MODULE: THE GENETIC ALGORITHM (GA) USED IN SAMUEL

The standard generational GA first *initializes* and then *evaluates* all population members. After these two steps, the GA enters a generational cycle: population members are *selected* for reproduction, copies of relatively fit individuals are *recombined* and *mutated*, and the new population members are *evaluated*, thus setting up the cycle for the next generation. Figure 3 gives pseudo-code for the standard GA.

```
begin
    t = 0;
    initialize P(t)
    evaluate structures in P(t)
    while termination condition not satisfied do
    begin
        t = t + 1;
        select P(t) from P(t - 1)
        crossover structures in P(t)
        mutate structures in P(t)
        evaluate structures in P(t)
    end
end
```

Figure 3. The Standard Genetic Algorithm (GA)

A user can find a detailed discussion of GAs in (De Jong, 1975; Holland, 1975; Goldberg, 1989). Several differences exist between the standard GA as implemented in GENESIS and the GA used in SAMUEL.

SAMUEL has several learning operators in addition to *mutation* and *crossover*. These operators are discussed in the sections below. Figure 4 gives the pseudo-code for the GA of SAMUEL.

```
begin
    t = 0;
    initialize P(t)
    select operators for P(t)
    evaluate structures in P(t)
    specialize (rules in) P(t) structures
    generalize (rules in) P(t) structures
    cluster (rules in) P(t) structures
    while termination condition not satisfied do
    begin
        t = t + 1;
        select P(t) from P(t - 1)
        select operators for P(t)
        crossover (on rule boundaries) structures in P(t)
        mutate (rules in) P(t) structures
        creep (rules in) P(t) structures
        merge (rules in) P(t) structures
        delete (rules from) P(t) structures
        update operator probabilities of P(t) structures
        evaluate structures in P(t)
        specialize (rules in) P(t) structures
        generalize (rules in) P(t) structures
        cluster (rules in) P(t) structures
    end
end
```

Figure 4. The Genetic Algorithm (GA) of SAMUEL

INITIALIZATION OF THE POPULATION

The standard GA typically starts with a randomly generated initial population to ensure an initially unbiased sampling of the search space. Instead, Samuel generates new rules for a plan based on experience (Schultz, 1990). There are three options for the specifying the plans of the initial population:

1. The plans may consist of one general rule, called a *maximally general rule*, that matches all sensor conditions (option 0),

2. All plans may have initially the same user-specified rules read in to SAMUEL from the **init** file (option 1), or
3. The initial plans may consist of a combination of user-specified rules from the **init** file and a maximally general rule (option 2).

The maximally general rule is internally defined in SAMUEL; that is, a user does not have to define a maximally general rule in the **init** file. In version 1.3 of SAMUEL, a user can specify only one set of rules at a time. In other words, SAMUEL only reads in one **init** file. (The version of SAMUEL that excludes the GA reads user-specified rules from the **rules** file instead of the **init** file; this version cannot be run without a **rules** file.)

A maximally general rule has the form

if (and) then (and).

Notice that this rule's condition part matches all sensor values and that the rule's action part specifies all action-values of all actions. A plan consisting of only this rule executes a random walk through the search space, since every action-value has equal strength. However, because the initial conditions for episodes are different due to random selection, the performance of plans having maximally general rules differ. After a successful episode, the *specialize* operator, discussed below, creates a new rule from a maximally general rule using sensor information. In this way, SAMUEL generates rules based on experience gained from successful episodes.

SELECTING OPERATORS

As mentioned in Chapter 3, each plan has an associated vector of operator probabilities. An operator probability is the independent probability of applying an operator in creating an offspring plan. (The probability vector does not store an operator probability distribution.) During operator selection, the GA considers each operator, one by one, for all plans of the population. If an operator probability is non-zero, the GA performs a Bernoulli trial to select whether or not to apply that operator. The GA records in a vector for each operator the indices of the plans using that operator. More than one operator may be used in forming an offspring plan. (Associating an operator probability vector with each plan supports the adaptive operator mechanism. See the section below on updating operator probabilities.)

EVALUATION OF EACH POPULATION MEMBER

As explained in the Introduction, the GA receives an evaluation of the plan from the CPS module in terms of the average payoff of a plan over a number of episodes. The GA uses this average payoff to determine the relative fitness of an individual plan over others in the population. Unlike the standard GA, the GA in SAMUEL scales the average payoffs so that selective pressure is maintained as the overall performance of the population rises each generation (Grefenstette, 1986). In SAMUEL, the *fitness* of each plan is defined as the difference between the *average payoff* received by the plan and a *baseline performance* measure. The baseline is the mean payoff received by the population minus one standard deviation from the mean. The baseline is adjusted slowly to provide a moderately consistent measure of fitness. Plans whose payoff fall below the baseline are assigned a fitness measure of zero, resulting in no offspring. This mechanism appears to provide a reasonable way to maintain consistent selective pressure toward higher performance.

SPECIALIZE, GENERALIZE, AND CLUSTER

Immediately after evaluating population members, SAMUEL applies the optional *specialize*, *generalize*, and *cluster* operators. The *specialize* and *generalize* operators are additional learning heuristics that SAMUEL uses to generate rules from experience. The *cluster* operator is logically associated with the *crossover* operator performed after *select*. Since, the rule firing information needed to apply the *cluster* operator is not available after the *select* operation, SAMUEL applies *cluster* after *generalize*.

One of the ways the GA creates new rules from existing ones is by applying the *specialize* and *generalize* operators. The *specialize* operator creates new rules by specializing rules having overly general conditions. The *generalize* operator creates new rules by generalizing rules having conditions that are too specific. These two operations differ from the *mutation* operator in that they add new rules to a plan rather than just modifying existing rules.

The Specialize Operator

In order to introduce plausible new rules, the GA applies to evaluated plans a rule modification operator called *specialize*. SAMUEL only applies *specialize* to a maximally general rule. The *specialize* operator is similar in spirit to Holland's *triggered operators* (Holland, 1986). The "trigger" in this case is the conjunction of the following conditions:

1. There is room in the plan for at least one more rule, and
2. A maximally general rule fired during a successful episode.

If these two conditions hold, *specialize* creates a new rule based on the sensor reading observed and the action taken on a given step of the episode. The condition for a sensor in the new rule covers approximately half the legal range for that sensor, splitting the difference (not necessarily equally) around the sensor reading. For example, suppose the initial plan contains the maximally general rule:

Rule 0: if (and) then (and)

Suppose further that the following step is recorded in the evaluation trace during the evaluation of this plan:

sensors: ... (range 500) (speed 1000) ...
action: (turn 0)
rule fired: Rule 6

Assuming that (1) the *range* sensor can assume values between 0 and 1500, (2) the *speed* sensor can assume values between 0 and 1400, and (3) the *turn* control can assume values between -180 and 180, then *specialize* would create the following new rule:

Rule 1:
if (and ... (range 250 1000) (speed 500 1200)
then (and (turn -90 90))

The GA gives this new rule a high initial strength when it is added to the plan. The maximally general rule remains in the plan at a lower rule strength. Rule 1 is plausible, since its action is known to be successful in at least one situation where the sensor values match the conditions on the left hand side.

The Generalize Operator

If a high performing rule only has partially matching conditions, the *generalize* operator creates a new rule so that boundaries on conditions match all the experienced sensor values. SAMUEL applies *generalize* to all rules except a maximally general rule.

The Cluster Operator

The *cluster* operator tries to arrange rules by their firing sequence using the best episodes from the last evaluation of the plan. Clustering of rules affects the operation of the *crossover* operator (as explained below).

SELECTION OF PLANS FOR CLONING

The first step in creating plans for the next generation is to select plans for copying or "cloning." Using fitness values, the GA selects for cloning those plans having relatively high fitnesses. High performing plans contribute more clones than low performing plans.

RECOMBINATION OF PLANS USING CROSSOVER

The select operator alone merely produces clones of high performance plans. In SAMUEL, a GA recombination operator called *crossover* works in concert with *select* to create plausible new plans. The GA moves through the population, applying *crossover* to randomly chosen pairs of clones. The *crossover* operator exchanges the rules of two plans to form two new offspring plans.

The Crossover Operator

The *crossover* operator in SAMUEL differs from the *crossover* operator usually applied in the standard GA. Unlike the 2-point *crossover* operator used in the standard GA, SAMUEL either applies a simple uniform *crossover* operator on rule boundaries if the plan's rules have not been clustered; otherwise, SAMUEL applies a restricted form of uniform *crossover*. Simple uniform *crossover* assigns each rule to one of the children plans with equal probability. When applying the restricted form of uniform *crossover*, SAMUEL assigns a sequence of rules that fired in succession during a successful episode to one of the two children with equal probability. In either case, a particular rule can only be assigned to one of the offspring. For example, to illustrate restricted *crossover*, suppose that the most recent traces of the parent plans are as follows:

Episodes 8 and 9 for Parent #1:

8.	$R_{1,3} \rightarrow R_{1,1} \rightarrow R_{1,7} \rightarrow R_{1,5}$	Successful Maneuver
9.	$R_{1,2} \rightarrow R_{1,8} \rightarrow R_{1,4}$	Failure

Episodes 4 and 5 for Parent #2:

4.	$R_{2,7} \rightarrow R_{2,5}$	Failure
5.	$R_{2,6} \rightarrow R_{2,2} \rightarrow R_{2,4}$	Successful Maneuver

One possible offspring might be

Offspring:

$$(\ldots R_{1,3} \; R_{1,1} \; R_{1,7} \; R_{1,5} \ldots R_{2,6} \; R_{2,2} \; R_{2,4} \ldots),$$

where the subscripts on the rules indicate the plan index and the plan's rule index, respectively.

Clustering rules ensures that a rule sequence achieving a successful maneuver is treated as a group during recombination. In this way, the offspring plans will likely inherit some of the beneficial behavior patterns of their parents. Of course, the success of any new combination of rules depends on the context provided by all the other rules in the plan.

MUTATION OF PLANS

After forming new plans using *crossover*, the *mutation* operators replace selected rules within the plans with modified versions of the existing rules. However, if a rule is marked as *fixed*, the rule is not subject to the mutation operators. (The user can specify a rule as *fixed* during program initialization. See Appendix C for details). There are two *mutation* operators in

SAMUEL: uniform *mutation* and *creep*.

The Uniform Mutation Operator

The *mutation* operator makes uniformly random changes to rules (and thus also plans). For example, *mutation* might alter a condition within a rule from

(range 500 1000)
to
(range 500 800).

Alternatively, *mutation* might change the action from

(turn 45 90)
to
(range -90 90).

The Creep Operator

The *creep* mutation operator makes a change that is restricted to the smallest possible change in a given atom. For example, if "speed" is an attribute with a granularity of 50, then *creep* might change an atom from

(speed 200 500)
to
(speed 250 500), or (speed 150 500), or (speed 200 450), or (speed 200 550).

THE MERGING AND DELETING OF RULES IN A PLAN

After the GA forms the new generation's plans, the GA may apply the *merge* and *delete* operators. These heuristics can be thought of as more generalized versions of mutation due to insertion and deletion. However, unlike the *mutation* and *creep*, the *merge* operator does not alter an existing rule. The *merge* operator introduces a new rule to a plan by combining information from existing rules. The *delete* removes a rule from the plan.

The Merge Operator

If two rules have the same actions (i.e., the same right hand side) then the *merge* operator creates a new rule by taking the union of the two rules' conditions (i.e., the unions of the rules' left hand sides). The *merge* operator does not include the newly formed rule in the plan if it intersects with a rule already in the plan having a different right hand side. As a result, the *merge* operator conservatively introduces rules into a plan that do not compete with the plan's existing rules.

The Delete Operator

The *delete* operator removes rules from a plan that are either subsumed by a better performing rule or that have very low activity.

UPDATING OPERATOR PROBABILITIES OF THE PLAN

Built into SAMUEL is a mechanism for adaptively updating operator probabilities, depending on the value of the **Op_update_rate** variable. If **Op_update_rate** is zero, then operator probabilities remain unaltered throughout a program run. This setting effectively produces the same result of operator application in the standard GA where the same operator probabilities apply to all plans. Setting **Op_update_rate** to a value between 0 and 1 activates the adaptive operator adjustment mechanism.

In the standard GA, the "genetic code" of a population member encodes information directly evaluated by the environment. By appending an associated vector of operator probabilities to population plans, the GA of SAMUEL also includes in the plan's genetic code information not directly evaluated. In SAMUEL, operator probabilities change as a by-product of the selection mechanism. If a new plan is generated using an operator, then the adjustment mechanism increases that operator's probability. Each generation, the GA adjusts a plan's operator probabilities as follows:

1. The GA systematically decays all of the operator probabilities associated with each plan by

$$1 - Op\_update\_rate.$$

2. If a new plan is generated using an operator, the new plan inherits the operator probabilities of its parent. The GA then increases (the previously decreased) operator probabilities of the new plan as follows:

$$prob = prob + Op\_update\_rate/(1 - Op\_update\_rate)$$

This approach takes advantage of the selection pressure already occurring within the genetic algorithm. SAMUEL implicitly evaluates operator probability levels along with other genetic information. As a result, the operator parameter adjustment used within the SAMUEL system does not require an explicit evaluation of operator fitness. Operator probabilities increase due to the fact that the offspring being generated represent the success of applying those operators.

Future studies of SAMUEL will address alternative ways of adaptively adjusting operator probabilities.

CHAPTER 6

THE PROBLEM SPECIFIC MODULE

This chapter describes the Problem Specific Module in general terms and in terms of the EM model specifically. The chapter also discusses how to interface a Problem Specific Module to the Performance Module.

GENERAL DESCRIPTION

As stated in the Introduction, the Problem Specific Module consists of the *sensor* and *control* interfaces, a *world model*, and a *critic*. Before performing matching on rule conditions, the CPS must obtain information from the *sensor* interface of the Problem Specific Module. The sensor interface transforms the state information of the world model to a form that can be used by the CPS.

After the CPS performs matching and conflict resolution to select a winning (fired) rule, the *control* interface of the Problem Specific Module converts the action-value expressed in that rule to a value that can be used by the world model. The control interface then communicates this transformed action-value to the *world model*. The world model updates the state of the environment.

The world model communicates new state information to the *critic* function so that the critic is able to evaluate the performance of the world model. The critic evaluates the performance at the completion of a task, or episode, in the world model. At that time, the critic communicates this evaluation to the Performance Module.

Upon receiving the critic's evaluation, the CPS performs credit assignment, updating the strengths of the rules actively involved in bidding to fire. After credit assignment, the Problem Specific Module records information from the CPS: in particular, the Problem Specific Module saves the CPS decision loop counter value. Before the CPS begins its next decision cycle, the world model communicates whether or not an episode has ended.

THE EM PROBLEM SPECIFIC MODULE

The Evasive Maneuver (EM) *world model* is an air combat simulation having two objects of interest: a plane and a missile. The objective of the problem is to control the turning rate of the plane to avoid being hit by the approaching missile. The missile can track the motion of the plane and steer itself toward the plane's anticipated position. The missile initially travels at a greater speed than the plane. However, the missile is less maneuverable since the missile has a greater turning radius than the plane and the missile gradually loses energy as it maneuvers. The simulation occurs over episodes that begin with the missile approaching the plane and that end when (1) the plane is hit, (2) the missile is exhausted, or (3) the time limit of the episode is exceeded.

The EM world model provides six *sensors* that give information about the current state:

1. *last-turn*: the current turning rate of the plane. This sensor can assume nine values, ranging from -180 degrees to 180 degrees in 45 degree increments.
2. *time*: a clock that indicates time since detection of the missile. Assumes integer values between 0 and 19.
3. *range*: the missile's current distance from the plane. Assumes values from 0 to 1500 in increments of 100.
4. *bearing*: the direction from the plane to the missile. Assumes integer values from 1 to 12. The bearing is expressed in "clock terminology", in which *12 o'clock* denotes dead ahead of the plane, and *6 o'clock* denotes directly behind the plane.
5. *heading*: the missile's direction relative to the plane. Assumes values from 0 to 350 in increments of 10 degrees. A heading of 0 indicates that the missile is aimed directly at the plane's current position, whereas a heading of 180 means the missile is aimed directly away from the plane.
6. *speed*: the missile's current speed measured relative to the ground. Assumes values from 0 to 1000 in increments of 50.

The current EM model has a single *control*: the turning rate of the plane. There are nine possible action-values: the turning rate can be set between -180 and 180 degrees, in 45 degree increments.

The *critic* provides payoff information at the end of each episode, defined by the formula:

payoff = 1000 if plane escapes missile.
= $10t+substeps$ if plane is hit at time t.

Ten *substeps* make up a decision step in EM.

IMPLEMENTING A WORLD MODEL

The sensor and control interfaces, the world simulation model, and the critic correspond to different parts of the SAMUEL system.

In setting up the sensor and control interfaces, a user needs to define the components of the attributes: the *name*, *type*, and *range*. These components need to be placed in the **attributes** file so SAMUEL knows how to store and interpret sensor and control information. (Chapter 2 discusses the knowledge representation of rules; Appendix B illustrates the syntax used in the **attributes** file.) The **read_sensors()** and **set_action(int action, int op)** procedures implement the sensor and control interfaces, respectively.

A user implements the heart of the *world model* with the **take_action**() procedure.

A user implements the critic function through the combined effect of the **take_action**() procedure and **double get_reward**() function. The **take_action()** procedure maintains the state variables that are used by **double get_reward()** to compute a reward.

The **init_env(unsigned int seed)** and **reset_env()** procedures perform between-plan and between-episode initialization of the environment, respectively. The **int end_of_episode**() function and **update_environment()** procedure ensure the coordination of the CPS's decision loop with the world model.

PROCEDURES AND FUNCTIONS

A user's implemented world model must contain the following procedures and functions: (1) **init_env(unsigned int seed)**, (2) **reset_env(), int end_of_episode()**, (3) **read_sensors**(), (4) **set_action(int action, int op)**, (5) **take_action()**, (6) **double get_reward**(), and (7) **update_environment**(). Procedure **cps()** calls all of these procedures and functions except for **init_env()**; procedure **eval()** (in **eval.c**) calls **init_env()**. (Procedure **main()** in **run-plan.c** (a program driver that excludes the GA) also calls **init_env()**). A user's **read_sensors()** procedure must call the CPS procedure **record_sensor(int i, int ptr_type, void *ptr)**. Notice that all of CPS's variables begin with capital letters.

init_env(unsigned int seed)

The **init_env(unsigned int seed)** procedure initializes world model variables between evaluations of plans.

In the EM model, **init_env(unsigned int seed)** clears the variables **episodes**, **successes**, **totalreward**, **winrate**, etc.

reset_env()

The **reset_env**() procedure initializes world model variables between episodes. The procedure also sets up the initial conditions in the world model.

In the EM model, **reset_env**() clears **endflag**, **hit**, and **substep**; the procedure also selects a random initial state for the plane and missile based on the reset parameters.

int end_of_episode()

The **int end_of_episode()** function performs end-of-episode clean up operations. The function returns 1 at the end of an episode, and 0 otherwise.

read_sensors()

The **read_sensors()** procedure transforms the internal state variables of the world model to sensor values used by **cps()**. The procedure may go through several steps to accomplish this transformation. Procedure **read_sensors()** calls the CPS procedure **record_sensor(int i, int ptr_type, void *ptr)** to store sensor values into **Sensor[i]**, an array common to EM and the CPS's (where **i** is the sensor index; **ptr_type** is the type of sensor, and **ptr** points to the sensor's value).

In the EM model, **read_sensors()** computes the distance, bearing, and heading of the missile relative to the plane. The procedure then converts these values into sensor readings with the possible addition of noise.

set_action(int action, int op)

Given the control action's index number (in **action**) and value (in **op**), **set_action(int action, int op)** converts the action-value specified in the fired the CPS rule to a form used in the world model.

In the EM model, the CPS expresses turn values using an index having the range [0, 9]; **set_action(int action, int op)** converts this value to an index having the range [-4, 4]. Subsequently, this turn index is translated into a real valued angle change of the plane in **take_action()**.

take_action()

The **take_action()** procedure simulates the effect of the control action in the world model.

In the EM model, the **take_action()** procedure also maintains state flags indicating how, and whether or not, an episode has ended. During the piloting of the plane, each decision step in actually requires several smaller time steps to accomplish. The world model's variable **substep** maintains the count of these time steps. The EM state variable **hit** indicates how an episode has ended; **endflag** indicates whether or not an episode has ended. The **endflag** is set when one of three conditions is met: (1) **hit** is set, (2) the missile's speed is below a minimum threshold, or (3) the number of decision steps (EM variable **step**) exceeds the maximum number permitted (**Maxtime**).

double get_reward()

Even though **cps()** calls **double get_reward()** every decision step, the function only returns a reward value at the end of an episode (i.e., when the **endflag** is set). Function **double get_reward()** calculates **reward** based on the state variables defined in **take_action()**.

In the EM model, the reward depends on the state variables **hit**, **step**, and **substep**. If the plane is hit, **reward** is (10)**step** + **substep**; otherwise, **reward** is 1000.

update_environment()

The **cps()** procedure calls **update_environment()** to update the world model's environment. At a minimum, **update_environment** increments the world model's decision step counter, **step**.

CHAPTER 7

DEMONSTRATION OF SAMUEL

SAMUEL is distributed as a tar file. From this tar file, a user creates a **./src** directory and a README file. The **./src** directory houses the source code of an archive library containing most of the functionality of SAMUEL. By using a special make target in the **./src** directory, a user can create additional working directories, separate from **./src**, for running experiments and for testing new world models. The user can also create a **./demo** directory.

INSTALLING SAMUEL

Perform the following steps to install SAMUEL:

1. Move the **samuel.tar** file to a directory where SAMUEL is to reside, and use the UNIX **tar** command on the **samuel.tar** file to create the **README** file and **./src** directory. On the command line, enter

   ```
   % tar -xf samuel.tar
   ```
2. Move to the **./src** directory, and edit the **makefile** to correctly set the **CFLAGS** and **EXTRA_LIBS** macros, depending on the type of system (Sun3, Sun4, or the Butterfly's Mach Uniform System). The default **CFLAGS** (-O4) invokes the optimizing compiler. The **EXTRA_LIBS** only needs to be defined when compiling for the Butterfly. The targets creating SAMUEL's executables already include the math library (-lm).
3. To create a demonstration directory called **./demo** in the directory where SAMUEL resides as well as to create the random archive library **libsam.a** in **./src** enter at the command line:

   ```
   % make all
   ```

RUNNING THE SAMUEL DEMONSTRATION

Change to the **./demo** directory:

```
% cd ../demo
```

The **demo** script in the **./demo** directory runs a shell script demonstrating the display output of SAMUEL for different plans. The script first copies one of the rule files (i.e., **rules.*name***) to the **rule** file used by SAMUEL during input, and then executes a version of SAMUEL that does not use the GA. Notice that the **params.demo** file does not have any GA parameters.

The **./demo** directory contains two files containing possible plans: **rule.best** and **rules.rand**. The **rules.best** file demonstrates graphically a high performing plan learned by SAMUEL in the EM environment. For basis of comparison, the **rules.rand** shows SAMUEL's performance in the same environment given a plan having rules with random action responses.

To run the SAMUEL demonstration, select one of the two rule files and enter the suffix of the rules file (i.e., best or rand) at the command prompt. For example, to create a display using the **rules.best** enter:

```
% demo best
```

To exit the demonstration, enter control C at the keyboard.

CREATING WORKING DIRECTORIES

The makefile in the **./src** directory provides a user with a three step procedure for creating versions of SAMUEL without altering the original source code. A working directory can contain any user-generated code: modified copies of the SAMUEL source code, new source code, and executables. Ideally, a user should create a separate working directory for each major set of experiments.

1. By selecting a working directory name (*directory_name*), and entering at the command prompt:

   ```
   %make exp "EXPDIR=directory_name"
   ```

a user can create a working directory. Notice that working directories must be created from the **./src** directory. After making a working directory, the system should respond with:

```
Created directory ../directory_name
Now cd to that directory and type...
  "make all"
```

2. Change to the working directory:

   ```
   % cd ../directory_name
   ```

   The working directory should contain the input files **attributes**, **rules** (for the **run-plan** executable), and **params**. The directory should also contain **params.def**, an on-line documentation file describing the parameters. (The directory does not contain a **init** file for running with **samuel.**) The "**make exp**" target should have also copied over from **./src** various shell scripts: (1) the **run-samuel** script and its children scripts **mkgraph**, **avegraph**, **getindex**, and **ttest**, and (2) **ch** and **p**, utility scripts for modifying and examining the **params** file, (3) **wins**, the utility script for summarizing the trace file, and (4) the **show** script for viewing one or two graphs using the **eview** previewer.

3. To make all of the SAMUEL executables in the working directory enter at the command line:

   % make all

   This creates the executables: (1) **samuel**, SAMUEL using the EM domain, (2) **run-plan**, a separate driver that allows a user to investigate the the EM domain using the CPS without having new plans generated by the GA, and (3) a couple of programs, **retest** and **extract**, used in the **run-samuel** script. See the section below on using the **run-samuel** shell script.

   If the working directory does not contain source code having the same names as the original source code in the **./src** directory, then issuing the make command creates the SAMUEL distribution executables. Otherwise, the compilation overrides the original distribution source code with user-defined versions. For example, suppose a user creates a working directory called **./EM_test1** and places a new version of **cross.c** in this directory to test the effectiveness of a new crossover operator in the EM environment. During compilation of **samuel**, the user's version of **cross.c** in **./EM_test1** supersedes the **cross.c** file used in making the **libsam.a** archive library.

MAINTAINING INPUT FILES

In general, to run SAMUEL, a user needs to consider three input files: **attributes**, **params**, and **init**. If a user runs SAMUEL without the GA, then the rule specification file is **rules** instead of **init**. SAMUEL expects to read files having these names. However, the default names **init** and **rules** can be changed by initializing the parameters **initfile** and **rulefile**, respectively. (See Appendix A1.)

The **attributes** file describes sensor and control attributes for the EM world model. Since an **attributes** file is domain-specific, a user typically has one **attributes** file for a given world model. Appendix B describes the **attributes** file in more detail.

If a user wants to examine SAMUEL's performance in solving a problem using different parameter scenarios, a user should maintain separate **params.*name*** files. Before a program run, a user should copy a particular file of interest to the **params** file.

Similarly, if a user plans to compare SAMUEL's performance for different initial plans, a user may have several rule initialization files. A user should put each plan to be tested in a separate file. Before running SAMUEL, a user would then copy a particular **init.*name*** file to **init**.

OUTPUT FILES OF SAMUEL

Depending upon user options, SAMUEL creates several output files. These files include: **best**, **detail**, **dump.env**, **last**, **log**, **out**, **rules.out**, **save**, **trace**, and **wins**. A user can change default names of the output files by setting the parameters in the **params** file. A user can also control which output files SAMUEL generates by setting flag parameters in the **params** file. Table 2 lists these flag parameters and the corresponding default output file names.

Flag Parameter	Default File Name	Module
etrace	**trace**	CPS
ruletrace	**rules.out**	CPS
best_interval	**best**	GA
last	**last.n**	GA
log	**log**	GA
out	**out**	GA
detail	**detail**	EM
dump	**"dump.env"**	EM

Table 2. Output Parameters

The GA parameter **best_interval** specifies the generational interval for writing out **best**; if **best_interval** is zero, SAMUEL does not create the file **best**.

SHELL SCRIPTS

The distribution includes some shell commands that make working with SAMUEL easier. These include: **ch**, **p**, **run-samuel**, **show**, and **wins**.

Modifying and Inspecting the params File

A user can alter parameters in the **params** file by using the **ch** command. For example to turn off the display, give the command:

```
% ch draw 0
```

The **ch** command first checks the **params.def** file to validate the parameter name before changing the **params** file. If a user does not specify a value for the parameter, then **ch** removes the parameter from the **params** file. For example,

```
%ch draw
```

removes parameter **draw** from the **params** file.

When a parameter is not included in the **params** file, SAMUEL assumes the default value for the variable associated the parameter. In this example, not having an entry in the **params** file for **draw** gives the same result as setting **draw** to zero, since the default value of **Drawflag**, the variable set by the **draw** parameter, is zero.

A user can display the current parameters in the **params** file by using the **p** command:

```
% p
```

Training and Testing Using run-samuel

In general, a plan should be tested a large number of episodes to obtain a good estimate of the plan's performance. Unfortunately, the computation time for evaluating plans is very long relative to the search time for discovering new plans. One technique for reducing the evaluation time is to decompose SAMUEL into a "training" component and a "testing" component. This decomposition technique is viable due to the fact that a GA performs well even when environmental feedback is approximate. During training, the CPS would provide the GA in SAMUEL with a rough average of a plan's performance using only a few episodes. Based on this performance estimate, the training system would filter out high performing plans for further evaluation. The "testing" system (identical to the CPS of SAMUEL) would then perform more extensive testing of these best performing plans asynchronously while the GA of SAMUEL continues to search for even better performing plans. The testing system in this scenario could represent an offline target system for the plans.

This decomposition technique is the motivation behind the **run-samuel** script. Instead of testing of the current best plans asynchronously, however, SAMUEL periodically collects over the generations some top percentage of the population's best performing plans into a "best" file. Each stored generation is called an *epoch*. SAMUEL generates a best file for each repetition of the experiment. After completing the training, a separate CPS module tests the plans in the best files using the same parameters as the training run, except that the the duration of the episodes is longer (e.g., 100) and the rule strengths of the plans are not adjusted.

The **run-samuel** script next retests extensively over even a larger number of episodes (e.g., 1000) the best plans from each epoch generated during the extended evaluation. If a user also specifies the suffixes of additional parameter files in the command line argument of **run-samuel**, then the script also retests the best plans of the epochs using those parameter files. Ultimately the **run-samuel** script generates files containing data for learning curve plots. A user can then plot a comparison graph showing the training run's average experimental learning curve and a particular testing run's average experimental learning curve.

Not only does the script provide a user with a convenient way to run SAMUEL, the script also provides a user with a easy way of testing the robustness of the run's best plans with respect to different sets of input parameters. For example, a user may want to investigate the robustness of a plan that has been trained without noise in the world model's sensors in a world model having noisy sensors. In this case, the training parameter list would have the **noise** set to zero (or not included, since the default is zero), and the testing parameter list would have **noise** set at some value between 0 and 1. If **noise** were set to 0.10, then a sensor would have added to it a random deviant from a normal distribution having a mean of 0.0 and a standard deviation equal to 10 percent of the legal range of that sensor.

The **run-samuel** script can take several arguments. The first argument is the suffix of a **params** file used as input for the training environment and the first testing environment. The remaining arguments are the suffixes of **params** files used as input to specify additional testing environments. If a user invokes **run-samuel** without arguments, then SAMUEL uses the **params** file for the training environment and the first testing environment. For example, suppose a user specifies three arguments: the first argument is **train**, the second argument is **noise1**, and the third argument is **noise2**. The command for running SAMUEL would be

```
% run-samuel train noise1 noise2
```

The **run-samuel** script would run SAMUEL using **params.train**, and test the resulting best files using **params.train**, **params.noise1**, and **params.noise2**. Notice that even though the parameter files used for testing may include GA parameters, GA parameters are not used during testing. The script tests plans without modification from the GA, and without modification from the CPS's credit assignment function. If **params.train** were used as a basis for training and testing, then **run-samuel** would generate three summary sets of data for plotting graphs: (1) learning curve **graph.train.train**, (2) learning curve **graph.train.noise1**, and (3) learning curve **graph.train.noise2**. A user would probably plot (1) and (2) together and (1) and (3) together to show comparisons.

Description of the run-samuel Script

The **run-samuel** script on-line contains useful comments. The description below elaborates on these comments. The beginning of the **run-samuel** script is as follows:

```
Comment   | Statement
----------+---------------------------------------------------
1.        | #!/bin/csh -f
          |
2.        | if ($#argv > 0) then
          |   cp params.$1 params
          | endif
          |
3.        | samuel
          |
4.        | set exps=`grep experiments params`
          | set exps=$exps[3]
          | set episodes=100
          | set ex=1
          |
5.        | while ($ex <= $exps)
          |    uncompress best.$ex
          |
6.        |    retest best.$ex $episodes | mkgraph > epoch.$ex
          |
7.        |    rm -f testplans.$ex
          |    set cmd='print $2'
          |    set n=`wc -l epoch.$ex`
          |    set n=$n[1]
          |    set j=1
          |    while ($j <= $n)
          |       cat epoch.$ex | awk "(NR==$j) {$cmd}"
          |              | extract -c best.$ex >> testplans.$ex
          |       @ j += 1
          |    end
          |
8.        |    rm epoch.$ex
          |    compress best.$ex
          |
9.        |    @ ex += 1
          | end
```

1. If a user removes the comment sign (#) from the first line, the **run-samuel** script works in subshell without reading the **.cshrc** file before invocation.
2. If there are arguments, the script copies **params** file of the first argument to **params**.
3. Using the first parameter file, the **run-samuel** script invokes **samuel**. For each experiment **ex**, where **ex** = 1,. . . , **experiments**, (**experiments** is specified in the **params** file), **samuel** creates a file **best.ex**. Two parameters affect the organization of each **best.ex** file: (1) **best_interval** sets variable **Best_interval** (default is 10), and (2) **bestpct** sets variable **Best_pct** (default is 0.2). **Best_interval** specifies the between-generation-interval, or epoch, for writing out the population's best plans to the **best.ex** file. **Best_pct** specifies the percentage of the population making up the best performing plans. Every **Best_interval** generations **samuel** appends the current **Best_pct** of the population to a **best.ex** file. Each **best.ex** is reduced in size using the UNIX **compress** command.
4. Before entering a while loop that examines each **best.ex** file, the script sets up some internal variables. The variable **ex** is the experiments loop counter. The variable **exps** is the number of experiments used as the upper bound condition in the while loop. The script isolates the number of experiments by using UNIX **grep** command to extract from the **params** file the expression containing the number of experiments (e.g., experiments = 10). The script sets the **episodes** variable, one of the command line inputs to the **retest** program invoked in the following while loop, to 100. To change the number of episodes for **retest**, a user would edit the **run-samuel** script.
5. After completing the **samuel** run, the script enters the while loop iterating over experiments. By using the UNIX **uncompress** command, the script first restores the **best.ex** file for the curent experiment so that **retest** can examine the file.
6. The **run-samuel** script runs **retest** in order to get a better statistical estimate of the performance of each plan. Using a **best.ex** file and **episodes** as command line input, the **retest** program recomputes the performance of the plans in the **best.ex** file using a larger number of episodes (e.g., 100) than the number used in the **samuel** run (e.g., 10). The **retest** gives all of the best plans an extended evaluation without modification by the GA or the CPS. Like the **samuel** program, the **retest** program uses the **params** file as input. However, **retest** overrides some parameters settings (without affecting **params**). Specifically, **retest** turns off the history mechanism (**History_flag** = 0), turns off the the profit sharing plan (**Psp_rate** = 0), and sets the maximum number of cycles (**Ncycles**) to 10,000,000. After processing a **best.ex** file, the script pipes the results of **retest** to the **mkgraph** script. The **mkgraph** script accepts as input the extended evaluation information from **retest**. For each epoch, the **mkgraph** script determines the best plan. Summaries of these best plans are then written to **epoch.ex**. This summary data include the index number of each epoch's best plan.
7. Using the index of each epoch's best plan stored in **epoch.ex**, the **extract** program extracts the best plan of each epoch from **best.ex**. The **extract** program writes out these plans to

testplans.ex. The **testplans.ex** file is written in a compact rule format that gives only numeric information from the rules.

8. Having extracted the necessary information from **best.ex**, **run-samuel** reduces the size of the **best.ex** files once again to save disk space by using the UNIX **compress** utility.

9. At the bottom of the first while loop, **run-samuel** increments the experiment counter, **ex**. The while loop repeats until **ex** is greater than **exps**.

The middle of the **run-samuel** script is as follows:

```
Comment   | Statement
----------+--------------------------------
10.       |  set episodes = 1000
          |  if ($#argv > 0) then
          |     set i = $argv[1]
          |     set k = 1
          |  else
          |     set i = params
          |     set k = 0
          |  endif
          |
11.       |  while ($k <= $#argv)
          |     rm -f graph.ave
          |     rm -f graph.all
          |     if ($k > 0) then
          |       set j = $argv[$k]
          |       cp params.$j params
          |     else
          |       set j = params
          |     endif
```

10. The second half of the script starts by setting the script variable **episodes** to 1000. This indicates that each epoch's best plan will be retested for 1000 episodes. The script then initializes the variables **i** and **k**. The variable **i** stores the first argument of the script (a suffix to a **params** file). This **params** file specifies the parameters used as a base of comparison. The script then initializes the following while loop's index variable **k**. If there are no arguments to the script, then the script examines only the default **params** file having no suffix, and the following while loop executes only once.

11. Next the script enters the first of two while loops, one nested inside the other. The outer while loop iterates over the index to the argument list of the **run-samuel** script (**k**). Based on this index, the script sets **j** to the suffix of current **params** file being retested. The first time through the loop, **j** is the first argument itself. The script then copies the current

params.j to the file **params**. Notice the script overwrites the existing **params** file in the working directory. If there are no arguments, then the file suffix stored in **j** is called params to indicate that there is no specific file.

The last part of the **run-samuel** script is as follows:

```
Comment  | Statement
---------+--------------------------------------------------------
12.      |     set ex = 1
         |     while ($ex <= $exps)
         |        retest testplans.$ex $episodes > graph.$i.$j.$ex
         |
13.      |        avegraph $i.$j.$ex
         |
14.      |        cat graph.$i.$j.$ex | getindex |
         |              extract testplans.$ex > bestplan.$i.$j.$ex
         |        rm graph.$i.$j.$ex
         |        @ ex += 1
         |     end
         |
15.      |     mv graph.all graph.$i.$j.all
         |     mv graph.ave graph.$i.$j
         |
16.      |     if ($i != $j) then
         |       ttest graph.$i.$i graph.$i.$j > ttest.$i.$j
         |     endif
         |     @ k += 1
         |  end
```

12. A nested inner while loop examines each of the experiments, **ex**. The **retest** program reads in the parameters specified in the **params** file as well as accepting the command line arguments indicating the plans to be tested (**testplans.ex**) and the number of episodes (**episodes**). By executing the the two while loops, the script retests plans using all parameter sets specified through argument list (outer loop index) for all experiments (inner loop index). Each retest generates output to **graph.i.j.ex**, where **i** is the first parameter file argument, **j** is some other argument (suffix of first through last **params** files), and **ex** is the index of the experiment.
13. Next, the **avegraph** script does the following: (1) joins each **graph.i.j.ex** file into a column of a table stored in file **graph.all**, and (2) for each point in the learning curve (each row), computes the average and standard deviation of the performance over the experiments examined so far in the inner while loop, and (3) writes out the current average learning curve to **graph.ave**. Each time through the while loop, **avegraph** writes over the old **graph.ave**.

14. Using **getindex** script, **run-samuel** finds the index of the best plan from the **graph.i.j.ex** file. This index is piped into the **extract** program which writes out the corresponding plan to **bestplan.i.j.ex**.
15. Before examining the next **params** file argument (index **j**), **run-samuel** copies the **graph.all** and **graph.ave** for the current parameter arguments to **graph.i.j.all** and **graph.i.j.ave**, respectively.
16. For the cases where different arguments are being examined (e.g., **i** = first parameter argument and **j** = second parameter argument), **run-samuel** performs a student t-test between the data points of the two files and writes the result of the test to **ttest.i.j**.

An Example Run Using run-samuel

In this example, we examine how sensor noise affects SAMUEL's performance. The example is based on an experiment reported in (Grefenstette, 1990).

There are two parameter files given as arguments to **run-samuel**: **params.train**, and **params.noise**. The **params.train** file is as follows:

```
;
; CPS PARAMS
;
episodes = 20
;
; EM PARAMETERS
;
control = 1
draw = 0
;
; GA PARAMS
;
experiments = 10
gens = 100
popsize = 100
length = 32
m_rate = 1
c_rate = 1
hist = 1
cluster = 1
spec_rate = 1
init = 0
save = 0
last = 0
single_act = 1
```

The **params.noise** file is identical to **param.train**, except that there is an additional line in the EM parameters: **noise** = 0.10. (Note: A user needs approximately 3.5 Megabytes of disk space to run the following example. The clock run time is approximately 13 hours on a dedicated Sun SPARCSTATION.) Suppose we make the following run by entering:

```
% run-samuel train noise
```

The **samuel** executable would repeat the experiment 10 times. For each iteration, the GA would generate new plans for 100 generations; each generation, there would 100 plans in the population; each plan would have no more than 32 rules, and the CPS would find an average evaluation of each plan using 20 episodes. Notice that in this run we have turned off the operator update mechanism (**op_rate** is zero).

In order to initialize each plan to a maximally general rule, we set **init** to 0. New rules are generated primarily through the *specialize* operator, so **spec_rate** is 1. In order to create diversity, the *crossover* rate and the plan's *mutation* rate are also very high (**c_rate = 1**; **m_rate = 1**). (The mutation rate *in a plan* is the default value of 0.01, since **mu_rate** is not defined.) Rules are clustered before *crossover* is performed (**cluster = 1**). Since we are clustering rules, the history mechanism needs to be on (**hist = 1**). Setting **single_act** to 1 constrains *specialize* and *mutate* to create rules specifying only one action-value (e.g., (turn 45 45)).

Every 10 generations, SAMUEL writes out the current best plan to file **best.ex** (which is the default interval of generations), where **ex** is the experiment number. No savefiles are to be kept, so **save** is zero. Setting the EM parameter **control** to 1 indicates that actions are selected through rule competition in the CPS. A user does not affect the program run through display inputs.

In all other cases, SAMUEL assumes the default parameter values. (In the on-line version of the **params** file, the **gen_rate**, **del_rate**, **merge_rate**, **creep_rate**, **crp_rate**, **op_rate**, and **good_payoff** parameters are explicitly set to their default values.)

The **run-samuel** script would generate learning curves **graph.train.train** and **graph.train.noise**. If a user has **eview** previewer installed, then the user can view these graphs as follows:

```
% show graph.train.train graph.train.train
```

Each plot has vertical bars at the data points indicating the standard deviations of the measurements.

Figures 5 shows a comparison of these learning curves. Rather than using error bars to indicate the range of the measurements, the plot uses vertical dotted lines to indicate significant statistical differences between the curves.

LEARNING CURVE COMPARISON: NOISE-FREE SENSORS AND 10 PERCENT NOISE

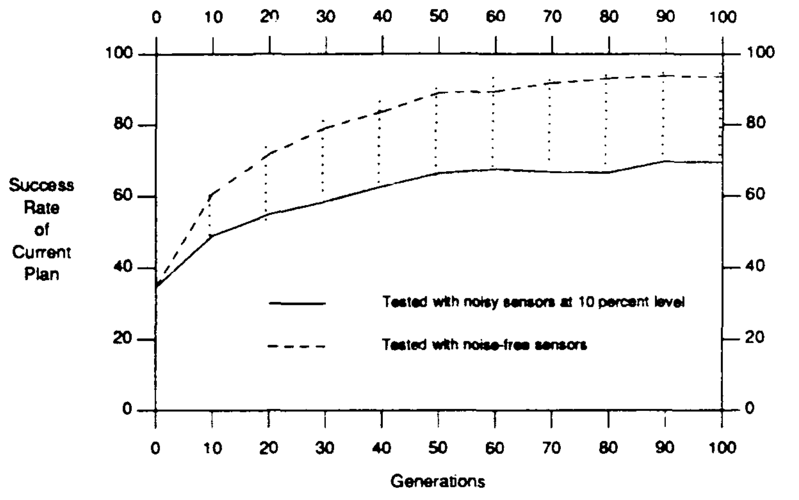


Figure 5. Learning Curves with Noise-Free Training Environment Testing at the 10 Percent Noise Level

Summarizing the Trace File

The **wins** command summarizes the **trace** file. The command

```
% wins trace
```

reports both mean and variance for the number of wins in samples consisting of 50 episodes each. The output format is:

```
totalwins  totalepisodes  mean  std  samples
```

WORKING DIRECTORIES FOR NEW PROBLEM SPECIFIC DOMAINS

The user should create a working directory for each new Problem Specific Module. For example, instead of creating a working directory for testing the EM environment (such as **./EM_test1**), suppose a user creates a new working directory called **./MYDOMAIN_test1**. A user should be aware that the **Makefile** makes two assumptions: (1) the **Makefile** expects a user to place the C source code modeling the Problem Specific Module in only one file (the EM domain uses the **em.c** file), and (2) the **DOMAIN** macro needs to be defined for each domain. For example, if a user's new model is in the **mydomain.c** file, then the user needs to set **DOMAIN = mydomain** in the **Makefile** of **./MYDOMAIN_test1**. If the user enters "**make all**" at the command line after modifying the **Makefile**, then the **MYDOMAIN_test1** directory should contain the executables **samuel** and **run-plan** that are specific to the new domain.

RUNNING SAMUEL STARTING FROM A CHECKPOINT FILE

The SAMUEL system provides a way for the user to periodically save the current state of the GA. Should a user's program stop before its normal completion, the user can restart SAMUEL based on information stored in a checkpoint file. To store a checkpoint file a user needs to define the following in the **params** file: **save_interval**, the number of generations between saving state information (default = 10), **save**, the number of save files maintained (default = 1), and **savefile**, the name of the save file (default = **save**). If there is more than one save file, then the GA saves the current state to these files in a round robbin fashion.

The GA saves its current state information by calling **save_state()** in procedure **generate()** before advancing the population pointers for the next generation. Procedure **save_state()** writes the GA's current state to the save file. To start the program again from the last saved state, a user needs to set the **restart** flag to 1 in the **params** file so that the **main()** program of **samuel** will call **restart()** during the initialization phase. Procedure **restart()** reads the file **save_state()** stored. If there is more than one savefile, the user also needs to specify, via parameter **savefile**, which savefile is to be read during the restart of **samuel**.

CHAPTER 8

PROGRAM SUPPORT AND MAINTENANCE

All questions and program bugs should be reported to:

John J. Grefenstette
Naval Research Laboratory, Code 5514
The Navy Center for Applied Research in Artificial Intelligence (NCAR A I)
4555 Overlook Ave., S.W.
Washington, D. C. 20375-5000

Internet: gref@aic.nrl.navy.mil
Telephone: (202) 767 - 2885

Suggestions and comments regarding improvements to this User's Manual are also welcome.

APPENDIX A: RUNTIME INPUT PARAMETERS

Many features of SAMUEL are controlled through run-time parameters. The **params** file contains the runtime parameter settings. The default value a parameter is in braces after the parameter name. The **params.def** file contains a table specifying the parameter's name, the associated variable name in SAMUEL, and the default initial value of the associated variable.

A user may list the parameters in the **params** file in any order. However, should the user specify a parameter multiple times, the parameter will take on the value of the last entry scanned.

A user may incorporate comments in the **params** file by beginning a line with ";" or "#."

In all cases where a file name is specified, the names "stdin" and "stdout" can be used instead to indicate that data are to be read from stdin, or printed to stdout, respectively.

APPENDIX A1: CPS RUNTIME PARAMETERS

Parameter	Variable	Default	Description
bbarate	**Bba_rate**	0.01	Update rate for bucket brigade.
cycles	**Ncycles**	1000000	Approximate upper bound for number of cycles to execute. The actual number of cycles may exceed this, because CPS always completes an episode once started. CPS runs until either cycles or episodes is reached at the beginning of an episode. A user can pick either one as the real limit by setting the other one high.
debug	**Debug**	0	If set, CPS prints voluminous tracing statements for debugging purposes.
episodes	**Nepisodes**	1000000	Upper bound on number of episodes to execute. CPS runs until either cycles or episodes is reached at the beginning of an episode. A user can pick either one as the real limit by setting the other one high.
etrace	**Etrace_flag**	0	If set, the CPS prints a line of statistics to **tracefile** for each episode, showing episode counter, number of steps, payoff, and number of partial matches.
hist	**History_flag**	1	Keeps an internal trace of each step performed by the CPS. This history is necessary for *specialize*, *cluster*, and *generalize* to work.

Parameter	Variable	Default	Description
outrulefile	**Outrulefile**	rules.out	Name of output file for **ruletrace**.
psprate	**Psp_rate**	0.01	Update rate for profit sharing plan.
randcr	**Rand_cr_flag**	0	If set, conflict resolution is "random" in the sense that the rule strengths are ignored. In fact, the first rule in the match set that suggests a given action becomes the bidder for that action, and all bids are considered equal.
rulefile	**Rulefile**	rules	Name of input file containing rules.
rules	**Nrules**	0	Number of rules in the **rule** file upon initialization.
ruletrace	**Rule_trace_flag**	0	If positive, prints out all the current rules to the **outrulefile** every **ruletrace** episodes.
seed	**Cps_seed**	987654321	CPS random number generator seed.
tracefile	**Tracefile**	trace	Name of output file for trace.

APPENDIX A2: GA RUNTIME PARAMETERS

Parameter	Variable	Default	Description
ancestors	**Ancestors**	2	Number of population arrays to keep. Need at least two, for current and last generation. Could be used for keeping genealogical traces.
base_rate	**Baseline_rate**	0.2	Rate at which the selection *baseline* is updated.
best_interval	**Best_interval**	10	Generations between printing to **bestfile**. If 0, no printing to **bestfile**.
bestfile	**Bestfile**	best	Output file for the best plans. Every **best_interval** generations, the top **bestpct** of the current population is printed in the **bestfile**.
bestpct	**Best_pct**	0.2	Percentage of population to print in the **bestfile**.
boost_trace	**Boost_op_trace**	0	Sends trace relating to operator adjustment of plans to standard out
c_rate	**Oprob[CROSS]**	0.0	Initial rate for *crossover*.
cluster	**Clusterflag**	1	If set, *crossover* assigns clusters of rules to the same offspring; otherwise, *crossover* performs uniform random crossover.

Parameter	Variable	Default	Description
cluster_rate	**Cluster_rate**	0.2	The inverse of the mean length of a cluster, where a cluster is a set of rules that fire in sequence in a successful episode. Used by *cluster* and *crossover* to assign clusters of rules to the same offspring.
creep_rate	**Oprob[CREEP]**	0.0	Initial rate for *creep*, which is like *mutation* but makes the smallest possible change instead of a random one.
crp_rate	**Creep_rate**	0.01	If positive, then **crp_rate** is the inverse of the mean waiting time between applications of creep. For example, if **crp_rate** = 0.01, the chance of any atom being mutated is about 1 percent.
del_rate	**Oprob[DEL]**	0.0	Initial rate for *delete* -- deletes inactive rules.
experiments	**Nexperiments**	1	How many experiments to run. Each experiment begins with a fresh initial population and a different random seed. If experiments > 1, the **bestfile** will have the experiment number appended to the names specified in the **params** file. For example if the **bestfile** is called **best** and **experiments** > 1, then the best plans for the first experiment will be printed in the file **best.1**.

Parameter	Variable	Default	Description
ga_seed	**Seed**	987654321	GA random number seed.
gen_rate	**Oprob[PCOV]**	0.0	Initial rate for *generalize* -- covers partially matched rule firings.
geneology	**Geneology**	0	If set, print out every plan evaluated by the GA. Could be used for genealogy studies, but takes up many megagbytes of storage.
gens	**Maxgens**	1000000	Maximum number of generations to run. GA runs until either **trials** or **gens** is reached at the beginning of an generation. A user can pick either one as the real limit by setting the other one high.
good_payoff	**Good_payoff**	1000.0	Payoff threshold for a successful episode. Used by *specialize*, *generalize*, and *cluster*.
init	**Initflag**	0	Plans are initialized depending on **init**. If: 0 with a general rule. 1 with rules from the **initfile** and a general rule. 2 with rules from **initfile**.
initfile	**Initfile**	init	Input file containing rule set used to seed initial population. See **init** for initialization method.
last	**Lastflag**	1	If set, save the final state of experiment *n* in file **last.***n*.

Parameter	Variable	Default	Description
length	**Length**	100	Maximum number of rules in any one plan. Maximum value: 512.
log	**Logflag**	1	If set, print the starting time, ending time, and overall offline, online, and best measure in the **logfile**.
max	**Maxflag**	1	If set, perform maximization; otherwise, perform minimization.
logfile	**Logfile**	log	File for printing system activity log.
m_rate	**Oprob[MU]**	0.0	Initial rate for *mutation*.
merge_rate	**Oprob[MRG]**	0.0	Initial rate for *merge* -- merges rules with same rhs, as long as new rule do not intersect with any rule already present.
mu_rate	**Mu_rate**	0.01	If positive, then **mu_rate** is the inverse of the mean waiting time between mutations. For example, if **mu_rate** = 0.01, the chance of any atom being mutated is about 1 percent.
op_rate	**Op_update_rate**	0.1	Rate for updating the operator probabilities. If 0, no updating is done.
out	**Outflag**	1	If set, writes out statistics each generation to the **outfile**.
outfile	**Outfile**	out	Output file for printing GA performance stats.

Parameter	Variable	Default	Description
popsize	**Popsize**	50	Population size.
restart	**Restartflag**	0	If set, read a previously saved state from the **savefile** and continue the run from that point.
savefile	**Savefile**	save	Output file for a snapshot of current state for later restarts.
save	**Nsaves**	1	Number of **savefiles** to keep.
save_interval	**Save_interval**	10	How often (in generations) to create a **savefile**.
single_act	**Single_act**	0	If set, *specialize*, *mutate*, and *creep* are constrained to create rules with a single value for each action, e.g., (turn 45 45) rather than (turn 0 90).
spec_rate	**Oprob[GCOV]**	0.0	Initial rate for *specialize* -- covers general rule firings.
stdev	**Stdev_weight**	1.0	The number of standard deviations *baseline* is below the mean. That is, if stdev = *n*, then *baseline* tends toward (mean - (*n*)(standard deviation)).
trace	**Traceflag**	0	If set, print tracing statements (usually, whenever a major GA function is entered or exited) to **tracefile**. For debugging.

Parameter	Variable	Default	Description
trials	**Maxtrials**	1000000	Approximate maximum number of *trials (evaluations) to perform.* The actual number of trials may exceed this, because GA always completes a generation once started. GA runs until either **trials** or **gens** is reached at the beginning of a generation. A user can pick either one as the real limit by setting the other one high.

APPENDIX A3: EM RUNTIME PARAMETERS

Parameter	Variable	Default	Description
bear_hi	**Bear_hi**	12.0	Upper bound for bearing at the start of each episode. Maximum **bear_hi** is 12.0.
bear_lo	**Bear_lo**	0.0	Lower bound for bearing at the start of each episode. Minimum **bear_lo** is 0.0.
control	**Control**	1	Indicates how actions are selected. If: 0 actions are random. 1 actions are selected by competition among the rules. 2 the user specifies the actions interactively. Interactive commands for EM: x — make a random turn s — straight r *n* — turn right 45*n* degrees l *n* — turn left 45*n* degrees q — terminate this episode (with success) *n* — turn (45*n* − 180) degrees d dumpfile — store env in named file c dumpfile — restore env from named file R — switch control to rules for the rest of the run z — take the action recommended by the firing rule

Parameter	Variable	Default	Description
delay	**Delay**	1	Controls the speed of the display.
detail	**Detail**	0	Controls level of detail printed in the **detailfile**. If: 0 no information is printed. 1 information concerning the sensor readings, the selected action and the firing rule is printed. 2 same as 1 except firing rule is not printed. 3 same as 2 except symbolic labels are omitted.
detailfile	**Detailfile**	detail	Indicates the name of the file to which detail information is printed.
draw	**Drawflag**	0	0 no display. 1 display with inertial background frame (both plane and missile move around screen). 2 display with plane centered (only missile appears to move) but orientation of background frame remains fixed. 3 display orientation is fixed with respect to plane -- this makes it easier to see the missile's relative motion wrt to the plane. 4 like 1 except uses a special font containing icons for plane and missile.

Parameter	Variable	Default	Description
dump	**Dump_env**	0	If **dump** > 0, the environment is stored in **dumpfile** when Cycle = **dump**. Can also be used interactively if **control** = 2 (and **draw** > 0).
dumpfile	**Dump_env_file**	"dump.env"	The file into which EM state variable are written during a dump.
em_seed	**Env_seed**	987654321	Seed for EM random number stream.
head_hi	**Head_hi**	0.0	Upper bound for bearing at the start of each episode. Maximum **head_hi** is 360.0.
head_lo	**Head_lo**	0.0	Lower bound for heading at the start of each episode. Minimum **head_lo** is 0.0.
maxtime	**Maxtime**	19	Controls the maximum number of steps in an episode. If the plane survives this long, it wins the episode.
mspeed_loss	**Mspeed_loss**	5.0	How much speed the missile loses when flying in a straight line.
mspeed_min	**Mspeed_min**	100.0	Missile speed threshold below which missile falls out of sky.
mturn_max_hi	**Mturn_max_hi**	PI/16	Upper bound for the maximum change in missile direction per substep.
mturn_max_lo	**Mturn_max_lo**	PI/16	Lower bound for the maximum change in missile direction per substep.

Parameter	Variable	Default	Description
noise	**Noise**	0.0	Controls noise to be added to sensors (before discretizing). If a sensor has a range of R, and its true value is X, the noisy value is selected from a normal distribution with mean X and standard deviation $(noise)(R)$.
pspeed_max	**Pspeed_max**	333.0	Plane speed when flying straight. The plane speed when turning is a decreasing function if the turning rate.
range_hi	**Range_hi**	1000.0	Upper bound for range at the start of each episode. Maximum **range_hi** is 1500.
range_lo	**Range_lo**	1000.0	Lower bound for range at the start of each episode. Minimum **range_lo** is 0.
restore	**Restore_env**	0	If **restore** > 0, the environmental state is restored from **restorefile** during the first cycle of the run.
restorefile	**Restore_env_file**	"dump.env"	The file from which EM state variables are read during a restore operation.
safe	**Safe_radius**	10.0	Radius of safe distance from plane to missile. If the closest point of approach drops below the safe range during any substep, the plane is hit.

Parameter	Variable	Default	Description
speed_hi	**Speed_hi**	700.0	Upper bound for speed at the start of each episode. Must be non-negative.
speed_lo	**Speed_lo**	700.0	Lower bound for speed at the start of each episode. Must be non-negative.
time_hi	**Time_hi**	0	Upper bound for time sensor at the start of each episode. Maximum **time_hi** is 19.
time_lo	**Time_lo**	0	Lower bound for time sensor at the start of each episode.
tstep	**Tstep**	10	Number of substeps per simulation step. During each substep, the plane and missile move in straight lines, and the screen is updated. Increasing this parameter improves the smoothness of the object's motion, but slows down the simulation.
turn_lo	**Turn_lo**	0	Lower bound for last-turn at the start of each episode. Minimum **turn_lo** is -180.
turn_hi	**Turn_hi**	0	Upper bound for last-turn at the start of each episode. Maximum **turn_hi** is 180.

Parameter	Variable	Default	Description
xscale	**Xscale**	40	Controls the screen magnification in the horizontal direction. Smaller values give greater magnification. Should be 80% of **yscale**.
yscale	**Yscale**	50	Controls the screen magnification in the vertical direction. Smaller values give greater magnification. Should be 125% of **xscale**.

APPENDIX B: THE ATTRIBUTES FILE

The **attributes** file describes the format of the rules. The first two lines must specify the number of conditions and actions

```
conditions = <n>
action = <m>
```

Each condition and action is then described in turn.
For linear or cyclic attributes, the format is:

```
condition <i>: (or action<i>:)
name = < cond-name >
type = linear (or cyclic)
low = < low-value >
high = < high-value >
values = < value-count >
```

where cond-name is a label of up to 16 characters, low-value and high-value are integers, and value-count is the number of endpoints in the range from low-value to high-value. For example, entries for some EM attributes are shown in Figure 6.

```
condition 3:
name = range
type = linear
low =   0
high = 1500
values = 16

condition 5:
name = heading
type = cyclic
low =  0
high = 350
values = 36

action 1:
name = turn
type = linear
low  = -180
high =  180
values = 9
```

Figure 6. EM **attributes** Entries.

For structured attributes the entry must define the tree of values bottom up. The format is:

```
condition <i>: (or action<i>:)
name = < cond-name >
type = structured
leaf-values = < n >
<value 1> ... <value n>
interior-values = < m >
name = < interior-value 1 >
children = < c_1 >
<child 1> ... <child c_1>
.
.
.
name = < interior-value m >
children = < c_m >
<child 1> ... <child c_m>
```

An example is shown in Figure 7.

```
condition 7:
name = weather
type = structured
leaf-values = 4
snow rain cloudy sunny
interior-values = 2
name = wet
children = 2
snow rain
name = dry
children = 2
sunny cloudy
```

Figure 7. Entry in **attribute** File for Structured Sensors.

For pattern attributes the format is:

```
condition <i>: (or action<i>:)
name = < cond-name >
type = pattern
pattern length = < l >
values = < n >
```

Here is an entry for a pattern attribute:

```
condition 8:
name = vision
type = pattern
pattern length = 6
values = 61
```

This defines `vision` as a 6 bit pattern that can take on binary values between 0 (000000) and 60 (111100), inclusive. For true bit patterns, the number of values will be a power of two, but this mechanism also allows using patterns for numeric attributes as well.

APPENDIX C: THE RULES FILE AND THE INIT FILE

Both the **rules** and **init** input files contain rules having the same syntax as those generated by SAMUEL on output. The **init** file contains initial rules used to seed SAMUEL's population. To use file input, the **init** parameter in the **params** file must be 1 or 2. The **rules** file is used in conjunction with running SAMUEL without the GA. (For example, **samuel** reads the **init** file; **run-plan** reads the file **rules**.) A rule in either file consists of four lines: (1) the rule number and the condition(s), (2) the action(s), (3) the parent, created, fixed, matched and partially-matched associated properties, and (4) the action, bid, active, fired, mean, var, strength, act, and bias associated properties. For example, a typical rule might be

```
Rule 1  if  (and (last-turn -90 135) (time 3 13) (range 200 1000)
               (bearing 5 11) (heading 270 200) (speed 50 450))
        then (and (turn 45 45))
        parent 0  created 0  fixed 1 matched 0  partially-matched 0
        action 1  bid 0  active 0  fired 0  mean 500  var  0 strength 500  act 0.5 bias 0
```

When a rule is *fixed*, it remains in plan without be replaced through modification during *mutation* or *creep*. The *fixed* designation only has significance when running **samuel**.

If an atom in a rule covers the entire range of a condition or action, then the atom does not have to be expressed in the rule. For example, in Rule 2, all of conditions are implicitly general except for the "last-turn" condition:

```
Rule 2  if  (and (last-turn -90 135))
        then (and (turn 45 45))
        parent 0  created 0  fixed 1 matched 0  partially-matched 0
        action 1  bid 0  active 0  fired 0  mean 500  var  0 strength 500  act 0.5 bias 0
```

The following rule expresses a random action for all sensor conditions:

```
Rule 3  if (and)
        then (and)
        parent 0  created 0  fixed 1 matched 0  partially-matched 0
        action 1  bid 0  active 0  fired 0  mean 500  var  0 strength 500  act 0.5 bias 0
```

Rule 3 is a maximally general rule. Notice that this is the same as the rule in **params.rand** in the **./demo** directory. This rule does not have to be explicitly included in the **init** file when running SAMUEL. Setting the **init** parameter to 1 in a **params** file ensures that SAMUEL uses both a maximally general rule and the rules from the **init** file.

APPENDIX D: FILES, FUNCTIONS, AND PROCEDURES

Appendix D gives implementation details of the SAMUEL system by file. The subappendices organize similar files into groups for easy reference. Appendix D1 lists files predominantly containing data structures. Most of the data structures for SAMUEL are in the **.h** files. For example, the complete definition of the **PLAN** data structure is in the **define.h** file; the complete definition of the **RULE** data structure is in the **common.h** file. Appendix D2 lists two driver files: **genesis.c**, the main driver for the SAMUEL system, and **run-plan.c**, a driver that permits the CPS to be run independently of the GA. Appendix D2 also lists the restart procedure in **restart.c** used for restart a SAMUEL run using a checkpoint file. Appendix D3 reviews the files making up the CPS module: the **attributes.c** and the **cps.c** files. Appendix C4 reviews the numerous files making up GA module: **cluster.c**, **creep.c**, **cross.c**, **delete.c**, **eval.c**, **evaluate.c**, **generalize.c**, **generate.c**, **init.c**, **measure.c**, **merge.c**, **mutate.c**, **ops.c**, **parallel.c**, **reset.c**, **select.c**, and **specialize.c**. Appendix D5 describes the files of the EM world model: **em.c** and **history.c**. The **em.c** file illustrates many of the procedures and functions that must be defined for the CPS interface. Appendix D6 lists files containing utilities. Some of these utilities are used by only one of the three modules. For example, the CPS uses the utilities in **atom.c**; the GA uses the utilities found in **best.c**. Appendix D7 lists the shell scripts included with this distribution. Appendix D8 lists the program input files.

APPENDIX D1: DATA STRUCTURE FILES

common.h

The **common.h** file contains data structures, constants, and macros common to the CPS, GA, and EM modules. The file contains the following constants pertaining to: (1) the pseudo-random generator, (2) upper bounds limits on arrays used in the file's type definitions, and (3) case statement descriptors. Type definitions for structures include **PARAM_TABLE**, **ATOM**, **RULE**, **ATTRIBUTE**, and **HISTORY**, as well as pointers to these structures. Macros include: (1) pseudo random generator (i.e., **RANDOM(SEED)**, uniformly random over [0, 1), **IRAND(SEEF, LOW, HIGH)**, uniformly random over an integer interval, and **URAND(SEED, LOW, HIGH)**, uniformly random over a real interval), (2) comparative operations (i.e., **MAX**, **MIN**) and (3) the **ABS** operation. File **common.h** includes the file **sets.h**.

cps.h and cps_extern.h

The **cps.h** file and its comparable external file **cps_extern.h** contain the definitions of constants and the declaration variables used by the CPS. The **cps.h** file contains the definition of the upper bound constants **SETS_PER_CONDITION** and **SET_SIZE**, used in the declaration of the *match set* arrays **Matchset[]** and **MS[]**. File **cps.h** sets default values for the variables associated with the input parameters of the **params** file. File **cps.h** also defines array **PARAM_TABLE cps_params[]** which sets up the correspondence between input parameters and several CPS variables. This correspondence is also documented in the **params.def** file. File **cps.h** includes **common.h**.

define.h

The **define.h** file defines constants and declares type structures for the GA. In particular, the file defines constants indicating the maximums for the number of ancestors saved in the history arrays (**MAX_ANCESTORS**), the population size (**MAX_POPSIZE**), and the number of operators (**MAX_OPERATORS**). The file defines mnemonics for each of the operators (**NOOP**, **MU**, **CROSS**, **SPEC**, **GEN**, **CREEP**, **DEL**, and **MRG**), as well as the default strength of newly generated rules, **DEFAULT_STRENGTH**.

The **define.h** file gives the type definition of a **PLAN**. In addition, **define.h** includes a output **TRACE** macro that prints out a string to standard output and flushes the buffer. File **define.h** includes **common.h**.

elog_defs.h

The **elog_defs.h** file contains definitions for the Mach Uniform System elog library

genesis.h and extern.h

The **genesis.h** and its comparable external file **extern.h** file declare variables used by the GA. These include: loop counters (**Experiment, Gen, Trials**), performance statistics (**Converged, Offline, Offsum, Online, Onsum, Stdev, Totbest, Totoffline**, and **Totonline**), performance save values (**Ave_current_value, Best, Best_individual, Best_current_value**, and **Worst_current_value**), variables saving initial values (**Initseed** and **Irules**), GA state variables (**Baseline, Best_next, Creep_next, Current_pop, Current_save, Mu_next**, and **Nextid**), and execution flags (**Doneflag**).

The **genesis.h** file also declares arrays and variables relating to operators: **Olist[MAX_OPERATORS][MAX_POPSIZE]**, the current offspring list, **Onext[MAX_OPERATORS]**, the current offspring count, **Oprob[MAX_OPERATORS]**, the initial operator probabilities, **V[MAX_OPERATORS]**, the current operator probabilities, and **op_pcnt[MAX_OPERATORS]**, the actual frequencies of operator application.

The **genesis.h** file sets the default values for the variables associated with the input parameters of the **params** file. In addition, the file defines array **PARAM_TABLE ga_params[]** which sets up the correspondence between the input parameters and several GA variables. This correspondence is also documented in the **params.def** file.

File **genesis.h** includes **define.h** and the standard **math.h** and **stdio.h** files. File **genesis.c** includes file **genesis.h**.

sets.h

The **sets.h** file include the type definitions of **unsigned int** SET_ELEMENT and its pointer.

APPENDIX D2: MAIN PROGRAM DRIVERS

run-plan.c

The **run-plan.c** file contains a version of procedure **main()** that runs EM without the GA. This main program first initializes the the sensor and control attributes by calling **get_attributes()**. Next, **main()** calls **get_params(cps_params)** and **get_params(env_params)** to initialize the CPS and EM parameters, respectively. If a a history of the plans is being maintained (i.e., the **History_flag** is set by initializing **hist** in the **params** file to be 1), then **main** calls **get_memory** to allocate memory for **HISTORY** structures. Enough memory is allocated to save rule information for each call **int eval(PLAN *p)** makes to **cps()**. The driver reads in a **rules** file. (The user can change the default name **rules** by setting the **rulefile** parameter in the **params** file.) The driver then initializes the CPS by calling **init_cps(unsigned int seed, int n)** and the environment by calling **init_env(unsigned int seed)**. Finally, **main()** calls **cps()** to run EM.

genesis.c

The **genesis.c** file contains the main program for SAMUEL. The procedure **main()** initializes the GA by calling **init_ga()**. If the **Logflag** is set, **main()** writes out the execution start time. If **Restartflag** is set, **main()** calls **restart()** to read in a previously saved state.

The main program then enters the experiments program loop. Nested within the experiments loop is the generations loop. Each generation, **main()** calls **generate** until **Doneflag** is set. After running an experiment, **main()** does the following: (1) accumulates performance measures, (by adding the current **Online**, **Offline**, and **Best** values into **Totonline**, **Totoffline**, and **Totbest** totals, respectively), (2) increments the counter **Experiment**, and (3) clears the generation counter **Gen** and **Doneflag**. The experiments loop ends when the counter **Experiment** is equal to the limit **Nexperiments**.

After exiting the experiments loop, **main()** computes the average online, offline, and best performances for the experiments. These results are written to the **log** file if **Logflag** is set.

restart.c

The **restart.c** file contains procedure **restart()**. If the **Restartflag** is set (the **restart** parameter in the **params** file is 1), then **main()** (in **genesis.c**) calls this procedure for restarting SAMUEL using state information stored in a checkpoint file (by default called **save**).

APPENDIX D3: CPS FILES

attributes.c

The **attributes.c** file contains the **get_attributes()** procedure. This procedure sets up data structures for sensors and controllers, indicating the ranges, the granularity, and the type of sensor or controller.

cps.c

File **cps.c** contains most of the procedures of the CPS.

init_cps(unsigned int seed, int n)

The **int eval(PLAN *p)** function (in **eval.c**) calls **init_cps(unsigned int seed, int n)** before calling the major procedure **cps()**. Initialization includes clearing the **Cycle, Episode**, and **Step** counters, setting the **Ave_Reward** variable to zero, and initializing the CPS random number generator seed, **Cps_seed**. Procedure **init_cps(unsigned int seed, int n)** also initializes variables relating to the set of active rules for each of the control actions (i.e., **Activesize[sn]**, **Firing[sn]**, **Fired[sn]**, **Firedsize[sn]**, **Fireingsize[sn]**, and **Mark[sn]**, where **sn** is the sensor number). Finally, the initialization procedure calls **compile_rules()** (in **rules.c**) which organizes the conditions of rules into a tree structure to facilitate the matching process.

*record_sensor(int i, int ptr_type, void *ptr)*

The **record_sensor(int i, int ptr_type, void *ptr)** procedure maps a string, integer, or a double precision real sensor value into an integer approximation. The **i** argument is the sensor index; the **ptr_type** argument indicates whether the sensor value is a string (**ptr_type** = STRING = 1), an integer (**ptr_type** = INT = 2), or double precision real (**ptr_type** = **DOUBLE** = 4), and **ptr** is a pointer to the sensor value. The procedure maps sensor values into both a discrete valued approximation of the sensor (stored in **Sensor[i]**) and an index corresponding to that approximation (**Index[i]**). The **record_sensor(int i, int ptr_type, void *ptr)** procedure first computes **Index[i]**); from this index the procedure then computes **Sensor[i]**.

cps()

The **cps()** procedure is the main execution loop for the CPS. After opening output files for traces (execution trace and rule trace), initializing the **payoff** variable to zero, and initializing the rule history data structures, **cps()** executes an episode loop lasting for **Nepisodes**, or for **Ncycles**, whichever comes first. These limits are compared to the **Episode** and **Cycle** counters, respectively. A cycle corresponds to a decision step within an episode. However, the **Cycle** counter is not reinitialized between episodes as is the decision **Step** counter. Thus, the main loop in **cps()** may curtail the number of episodes (to be less than **Nepisodes**) if the episodes tend to last for too many steps.

The decision loop is nested inside the episode loop. For each decision step, **cps()** calls several functions. At the top of the loop **cps()** calls **read_sensors()** (in **em.c**) to obtain sensor values in the world model. Next, **cps()** calls **find_matches()** to determine which rules have conditions that match sensor readings. For each control action, **cps()** calls **resolve_conflicts(int action)** followed by **set_action(int action, int op)**. The **resolve_conflicts(int action)** function selects an action-value among equally matching rules; **set_action(int action, int op)** (in **em.c**) converts an action-value to one understood by the controller in the environment. After selecting an action-value, **cps()** calls **take_action()** to actually carry out the control action in the environment. To get the critic's evaluation of the decision step (which is null until the end of the episode), **cps()** calls function **double get_reward()** and sets the result to the CPS variable **Reward**. Finally, **cps()** calls **update_strengths()** and **update_environment()** to modify the rule strengths and update world model, respectively.

After leaving the decision loop, **cps()** increments **payoff** by **Reward**, thus maintaining a sum of the payoffs over the current set of episodes.

update_strengths()

For each control action **ca**, the **update_strengths()**: (1) saves the current set of *active* rules by writing **Firing[ca]** to **Fired[ca]**, and (2) reinitializes **Firing[ca]** and the counter **Firingsize**[ca]. Finally, **update_strengths()** calls **psp()** to adjust rule strengths using the Profit Sharing Plan.

APPENDIX D4: GA FILES

cluster.c

The **cluster.c** file contains the **cluster(PLAN *p)** procedure. After calling **eval(PLAN *p)** to evaluate plan and then calling **specialize()** and **generalize()** to modify a plan, **evaluate()** (in **eval.c**) calls **cluster()**. If **Clusterflag** is set, **cluster(PLAN *p)** arranges rules together based on firing sequence using the best episodes in the last evaluation of the plan; otherwise, no clustering is performed. The **History** structure stores information on the last evaluation of the plan. Even with **Clusterflag** set, clustering only occurs if the **History.payoff[best plan]** is greater than or equal to **Good_payoff**. (**Good_payoff** is by default 1000.0; a user can specify a value for **Good_payoff** through the **good_payoff** parameter in the **params** file.) Clusters of rules are randomly assigned to the offspring. Stochastic selection from an exponential distribution having a mean **Cluster_rate** determines which offspring receives a cluster. All rules, whether fired or not, are assigned to one of the offspring. The subsequent crossover operation varies depending on whether or not rules have been clustered.

creep.c

The **creep.c** file contains a procedure and a related function: **creep()** and **char creep_plan(PLAN *pl)**.

The **CREEPLIST[plan index]** (the same as the **CREEP** vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *creep* operator. The **CREEPMAX** (the same as the **CREEP** dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

The **creep()** procedure calls **char creep_plan(PLAN *pl)** in an iterative loop for all plans in the CREEPLIST. Function **char creep_plan(PLAN *pl)** attempts to perform the *creep* operation on a plan **pl** and returns 1 or 0 depending the success of the operation. If **char creep_plan(PLAN *pl)** has changed the plan (i.e., **changed** is set), then **creep()** calls **boost_oprob(PLAN *p, int op)** (in **ops.c**) to increase the *creep* operator probability of that plan (assuming **Op_update_rate** is greater than zero). At the bottom of the loop, **creep()** stores the percent usage of the *creep* operator in the population in the **op_pcnt[MAX_OPERATORS]** array. Outside of the loop, if the **Boost_op_trace** flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not the plan was "creeped."

Function **char creep_plan(PLAN *pl)** applies the *creep* operator to both the conditions and actions of rules. The **Creep_next** counter saves the current position in the plan where the operator is being applied. The function updates **Creep_next** using stochastic selection from an exponential distribution having a mean of **1/Creep_rate**. After testing to make sure that a rule is not *fixed*, **char creep_plan(PLAN *pl)** calls **copy_rule(RULE *r, RULE *q)** (in **rules.c**) to copy the current rule into a temporary rule structure. The function **char creep_plan(PLAN *pl)** applies the *creep* operator to a condition or an action part of the rule by calling

creep_atom(ATTRIBUTE *attribute, ATOM *atom) (in **atoms.c**). If the function modifies a rule to one not already in the plan, **char creep_plan(PLAN *p1)** calls **copy_rule(RULE *r, RULE *q)** to copy the rule in the temporary rule structure back to the gene.

cross.c

The **cross.c** file contains a procedure and a related function: **crossover()** and **int crossover_plans(PLAN *p1, PLAN *p2)**.

The **CLIST[plan index]** (the same as the **CROSS** vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *crossover* operator. The **CMAX** (the same as the **CROSS** dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

The **crossover()** procedure calls **int crossover_plan(PLAN *p1, PLAN *p2)** in an iterative loop, passing to the function each time a pair of plans from the **CLIST**. Function **int crossover_plan(PLAN *p1, PLAN *p2)** performs uniform crossover of the parent plans on rule boundaries. The function returns the value 0 through 3, depending on which parent(s) contributed to making children: if neither parent contributed, then the returned value is 0; if only the first parent contributed, then the returned value is 1; if the second parent contributed, then the returned value is 2; if both parents contributed, then the returned value is 3. The **crossover()** procedure then calls **boost_if_changed(int changed, int mom, int dad, int opcode)** (in **ops.c**) which in turn calls **boost_oprob(PLAN *p, int op)** (also in **ops.c**) to increase the *crossover* operator probability of those plans contributing offspring (assuming **Op_update_rate** is greater than zero). (Procedure **boost_if_changed(int changed, int mom, int dad, int opcode)** also stores the percent usage of the *crossover* operator in the population in the **op_pcnt[MAX_OPERATORS]** array.) At the bottom of the loop, **crossover()** computes **Converged** based on the average percentage of duplicate rules in the plans.

Function **int crossover_plans(PLAN *p1, PLAN *p2)** goes through several steps to perform uniform crossover. In order to sense later on whether or not there is a new offspring, the function resets the flags **changed**, **mom_changed**, and **dad_changed**. Next, the function marks the "status" field of the rules of each parent to indicate whether or not a rule is the same in both parent plans (i.e., a *duplicate*). Before proceeding with *crossover*, the function makes sure that one parent does not subsume another. If one does subsume another, the function returns without performing *crossover*. After computing the average percentage of duplicate rules for a pair of plans, the actual *crossover* operation proceeds.

The *crossover* operation is done in three steps. First, the function **int crossover_plans(PLAN *p1, PLAN *p2)** copies all of the duplicate rules into both offspring. Second, the function copies rules from parent plans to the offspring plans based on the rule assignments determined in **cluster(PLAN *p)** (called previously from **evaluate()**). When **Clusterflag** is set, rules that have fired are preassigned to one of the offspring before calling **crossover()**. Third, the function randomly assigns unassigned rules to the offspring. A rule is unassigned when: (1) rule clustering is off, or (2) rule clustering is on, but the rule remains unassigned since it did not fire.

If the **Boost_op_trace** is set, the function writes an output trace indicating which plans created new offspring. After assigning rules to the offspring, **int crossover_plans(PLAN *p1, PLAN *p2)** overwrites the parent plans with the the offspring plans. Before returning control to **crossover(), int crossover_plans(PLAN *p1, PLAN *p2)** uses the **mom_changed** and **dad_changed** flags to encode the returned **changed** value.

delete.c

The **delete.c** file contains a procedure and a related function: **delete()** and **int delete_rules(RULE *rule, int length)**.

The **DELLIST[plan index]** (the same as the **DEL** vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *delete* operator. The **DELMAX** (the same as the **DEL** dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

The **delete()** procedure calls **int delete_rules(RULE *rule, int length)** in an iterative loop for all plans in the **DELLIST**. Function **int delete_rules(RULE *rule, int length)** goes through all of the rules in a plan, testing whether or not to apply the *delete* operator. When done, the function returns the remaining number of rules in the plan. If **int delete_rules(RULE *rule, int length)** has reduced the number of rules in the plan, then **delete()** calls **boost_oprob(PLAN *p, int op)** (in **ops.c**) to increase the *delete* operator probability of that plan (assuming **Op_update_rate** is greater than zero). At the bottom of the loop, **delete()** stores the percent usage of the *delete* operator in the population in the **op_pcnt[MAX_OPERATORS]** array, and saves the new length of the plan in the plan's length field. If the **Boost_op_trace** flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not rules were deleted from the current plan.

Function **int delete_rules(RULE *rule, int length)** examines for all action parts of the rules, all pairs of rules in the plan to see whether or not a rule can be deleted. Each rule's strength is tested against a threshold strength (stored in variable **threshold**) which is the maximum of 500 or 110 percent of one of the other rules' strength. If a rule's stength is greater than **threshold**, then the **stronger** flag is set to indicate that the rule exceeds the threshold. If the stronger rule also subsumes the other rule (tested by calling **subsumed(RULE *r, RULE *q)** (in **rules.c**)) and the rule is not fixed, then the function sets the subsumed rule's status to zero. A zero status marks the rule for deletion. Function **int delete_rules(RULE *rule, int length)** also deletes non-fixed rules having low activity. If a rule's activity is less than 0.01, a rule's status is set to zero. Rules having a non-zero status are copied back sequentially into the plan using **copy_rule(RULE *r, RULE *q)**.

eval.c

File **eval.c** contains the **int eval(PLAN *p)** function. The **int eval(PLAN *p)** function first translates the genetic structure of the passed plan (**RULE *gene** field in **PLAN**) into the rules needed by the CPS (**RULE**). Next, **int eval(PLAN *p)** initializes the following before calling the CPS: (1) the pseudo-random generator seed, (2) the CPS by calling **init_cps(unsigned int seed, int n)**, and (3) the EM environment by calling **init_env(unassigned in seed)**.

Function **int eval(PLAN *p)** then calls **cps()** to obtain the plan's average payoff per episode. The function stores one tenth of this average payoff to the plan's "value" field (*value* associated property). The difference between the average payoff and **Baseline** is stored in the plan's "fitness" field (*fitness* associated property). To save the current set of best plans (in **Bestset**) to a file, the function calls **savebest_plan(PLAN *p)**. Finally, **int eval(PLAN *p)** performs the inverse translation of copying the plan's rules (now modified in rule strength) back into a genetic structure used by the GA.

evaluate.c

The **evaluate.c** file contains the **evaluate()** procedure. Procedure **evaluate()** first stores the generation generation and the trial associated with each plan into the plan's "gen" and "trial" fields. For each member of the population, **evaluate()** calls: (1) **eval(PLAN *p)**, (2) **specialize(int i)**, (3) **generalize(int i)**, and (4) **cluster(PLAN *p)**. If the **US** compilation flag is set, **evaluate()** executes alternative code for performing parallel processing on the Butterfly.

generalize.c

The **generalize.c** file contains a procedure and a related function: **generalize(int i)** and **int generalize_plan(RULE *rule, int length)**.

The **GENLIST[plan index]** (the same as the **GEN** vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *generalize* operator. The **GENMAX** (the same as the **DEL** dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

Procedure **evaluate()** calls **generalize(int i)** within an iterative loop over population plans, passing the plan's index each time. This differs from the **crossover()** or **mutate()** procedures, where the iterative loop over plans is in the operator procedure itself. As a result, **generalize(int i)** first needs to find the plan's comparable **GENLIST** index given the plan's population index. The procedure saves the current number of rules in the plan (in **oldlength**) before calling function **generalize_plan(RULE *rule, int length)** to perform the actual generalize operation. This function returns the resulting number of rules in the plan, and **generalize(int i)** stores this number in **newlength**. At the bottom of the loop, if there is a change in the plan's length, **generalize(int i)** stores the percent usage of the **generalize** operator in the population in the **op_pcnt[MAX_OPERATORS]** array, and then saves **newlength** in the plan's length field. If the

Boost_op_trace flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not rules were generalized in the current plan.

Function **int generalize_plan(RULE *rule, int length)** examines the episodes associated with a plan. Starting with the first episode as a basis of comparison, the function performs a linear search for an episode that has a higher payoff at least as high as **Good_payoff** (set via the the **good_payoff** parameter in **params**). If the search is successful, the function attempts to generalize rules in the episode. This episode then becomes the new basis of comparison in finding an episode having an even higher payoff.

When **int generalize(RULE *rule, int length)** finds a better performing episode, the function tests each rule in the episode to see if it fired due to a partial match. If so, the function copies the fired rule into a new rule structure, and then examines the conditions of the clone rule to determine which ones need generalizing. If a condition does not match the sensor value experienced when the rule fired (recorded in the **History.sensors[decision step][condition index]**), then the function generalizes the condition atom of the clone rule by calling **generalize_atom(*attr, ATTRIBUTE *atom, unsigned int value)**.

After generalizing a rule, **int generalize(RULE *rule, int length)** initializes the generalized rule, indicating the rule's parent, creating time, and other associated properties. If the new rule is unique, then it is added to the plan.

generate.c

The **generate.c** file contains the **generate()** procedure. The **generate()** procedure is the main program loop for the GA. The main program, **main(int argc, char *argv[])**, calls **generate()**. For the first generation, **Gen** = 0, **generate()** calls **reset_ga()** to initialize the population and **select_ops()** to select the GA operators for each plan.

The procedure then calls **evaluate()** to obtain performance estimates of the initial plans, bypassing the generation of new plans using GA operators. After evaluating plans, the procedure performs end-of-loop tasks: (1) stores intermediate results by calling **measure()**; (2) sets the **Doneflag** when either the trial counter exceeds the maximum trials (**Trials >= Maxtrials**) or the generations counter exceeds the maximum generations (**Gen >= Maxgens**); (3) prints out the best performing plans by calling **printbest()**; (4) increments the generations counter, **Gen**; (5) saves the current state in a savefile by calling **save_state()**, and (6) advances the population pointers for the next generation by setting **Old** to **New**. The procedure then returns control to **main(int argc, char *argv[])**.

When **main(int argc, char *argv[])** calls **generate()** in subsequent generations, **generate()** starts by creating a new population of plans. As a first step, **generate()** calls **select()** to make copies of selected plans. The **select_ops()** then selects the operators each plan will use in creating new population members. To create new plans from these copies by applying the various operators **generate()** calls **crossover()**, **delete()**, **mutate()**, **creep()**, and **merge()**. Each of these operator procedures checks the **Olist[operator index][entry number]** array to locate those plans applying the operator in creating new offspring plans. The **generate()** procedure calls **update_oprobs()** to adjust the operator probabilities. (Procedure **update_oprobs()** only creates new plans if the operator update mechanism is on.)

Having created new plans, the **generate()** procedure once again calls **evaluate()**. The cycle repeats until **main(int argc, char *argv[])** no longer calls **generate()** due to **Doneflag** being set.

init.c

The **init.c** file contains the **init_ga()** procedure. The **init_ga()** procedure first resets the **Experiment** counter and the **Doneflag**. Next, the procedure reads in the **attributes** file by calling **get_attributes()** and the parameter file by calling **get_params(PARAM_TABLE *partab)** for each type of parameter (i.e., CPS parameters, environmental parameters, and GA parameters).

After reading in file inputs, **init_ga()** sets up several dynamic arrays. If the history flag **History_flag** is set, then **init_ga()** allocates memory for the history structure The **init_ga()** procedure also allocates memory for the **New** and **Old** population arrays and their gene structures and for the array of best plans (**Bestset**) and their gene structures. After **init_ga()** initializes memory for any initial rules, the procedure reads these rules into array **Initset**.

measure.c

The **measure.c** file contains the **measure()** procedure. The **measure()** procedure records several statistics of a population's performance including: **Best_current_value**, **Best_individual**, **Worst_current_value**, **Ave_current_value**, **Best**, **Onsum** (to compute **Online**), **Offsum** (to compute **Offline**), **maxlength**, **minlength**, **avelength**, **Stdev** (of the average current value), and **Baseline**. Variables **maxlength**, **minlength** and **avelength** are internal to **measure()**; the remaining variables are declared in **genesis.h**.

merge.c

The **merge.c** file contains a procedure and a related function: **merge()** and **int merge_rules_from_plan(PLAN *p, int length)**.

The **MRGLIST[plan index]** (the same as the **MRG** vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *merge* operator. The **MRGMAX** (the same as the **MRG** dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

The **merge()** procedure calls **int merge_rules_from_plan(PLAN *p, int length)** in an iterative loop for all plans in the **MRGLIST**. The function attempts to merge rules in a plan to form new rules for the plan. Function **int merge_rules_from_plan(PLAN *p, int length)** returns the number of rules in the resulting plan. If there are more rules after performing *merge*, then **merge()** calls **boost_oprob(PLAN *p, int op)** (in **ops.c**) to increase the *merge* operator probability of that plan (assuming **Op_update_rate** is greater than zero). At the bottom of the loop, **merge()** stores the percent usage of the *merge* operator in the population in the **op_pcnt[MAX_OPERATORS]** array. At the bottom of the loop, if the **Boost_op_trace** flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not the plan generated new rules using *merge.*

In a pairwise comparison of all rules in a plan, function **int merge_rules_from_plan(PLAN *p, int length)** calls **same_act(RULE *q, RULE *r)** to see if two rules have the same action-values. If so, then the function calls **rule_union(RULE *r, RULE *q, RULE *s)** to create a new rule by forming the union of the two rules' conditions. The **int merge_rules_from_plan(PLAN *p, int length)** rejects this new rule if (1) its conditions intersect with some other rule in the plan (determined by calling **cond_intersect(RULE *q, RULE *r)**) and (2) its action-values differ from this other rule. Otherwise, the function updates the rule index counter for the plan, initializes the rules' parent and creation time associated properties, and writes this new rule into the plan (using **copy_rule(RULE *r, RULE *q)** in **rules.c**).

mutate.c

The **mutate.c** file contains a procedure and a related function: **mutate()** and **int mutate_plan(PLAN *p)**. The structure of file **mutate.c** is the same as **creep.c**.

The **MULIST[plan index]** (the same as the MU vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *mutate* operator. The **MUMAX** (the same as the MU dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

The **mutate()** procedure calls **char mutate_plan(PLAN *pl)** in an iterative loop for all plans in the MULIST. Function **char mutate_plan(PLAN *pl)** attempts to perform the *mutate* operation on a plan **pl** and returns 1 or 0 depending on the success of the operation. If **char mutate_plan(PLAN *pl)** has changed the plan (i.e., **changed** is set), then **mutate()** calls **boost_oprob(PLAN *p, int op)** (in **ops.c**) to increase the *mutate* operator probability of that plan (assuming **Op_update_rate** is greater than zero). At the bottom of the loop, **mutate()** stores the percent usage of the *mutate* operator in the population in the **op_pcnt[MAX_OPERATORS]** array. Outside of the loop, if the **Boost_op_trace** flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not the plan was mutated.

Function **char mutate_plan(PLAN *pl)** applies the *mutate* operator to both the conditions and actions of rules. The **Mu_next** counter saves the current position in the plan where the operator is being applied. The function updates **Mu_next** using stochastic selection from an exponential distribution having a mean of **1/Mu_rate**. After testing to make sure that a rule is not *fixed*, **char mutate_plan(PLAN *pl)** calls **copy_rule(RULE *r, RULE *q)** (in **rules.c**) to copy the current rule into a temporary rule structure. The function **char mutate_plan(PLAN *pl)** applies the actual mutate operation on a condition or an action part of the rule by calling

mutate_atom(ATTRIBUTE *attribute, ATOM *atom) (in **atoms.c**). If the resulting *mutate* operation results in a rule not already in the plan, **char mutate_plan(PLAN *p1)** calls **copy_rule(RULE *r, RULE *q)** to copy the rule in the temporary rule structure back to the gene.

ops.c

The **ops.c** file contains procedures involving a plan's operator probability vector.

select_ops()

Based on the plan's operator probability vector, a plan selects the operators to be used in creating offspring plans.

Procedure **select_ops()** first initializes three arrays. The **Onext[operator index]** array stores the number of plans using an operator; the **V[operator index]** array stores the average probability of applying an operator over plans, and the **op_pcnt[operator index]** array stores the actual percent usage of the operator. The **select_ops()** procedure defines values for the first two arrays; each operator procedure computes **op_pcnt[operator index]** after applying the operator.

For each member of the population, **select_op()** selects a set of operators. In an iterative loop over operator indices, the function randomly selects whether or not to include the operator in the plan's operator set based on the operator's probability stored in the plan's operator probability vector. If a plan uses an operator, the plan's index is stored in the 2-dimensional array **Olist[operator index][Onext[operator index]]**.

boost_oprob(PLAN *p, int op)

If one of the GA operator procedures (e.g., **mutate**) successfully makes a change in a plan, then it calls **boost_oprob(PLAN *p, int op)**. If **Op_update_rate** is greater than zero, then **boost_oprob(PLAN *p, int op)** increments the probability of **op** in plan **p** by $Op\_update\_rate/(1 - Op\_update\_rate)$.

update_oprobs()

Every generation after applying the genetic operators, if **Op_update_rate** is greater than zero, **generate()** calls **update_oprobs()** to decay all of the operator probabilities in all of the plans by $1 - Op\_update\_rate$.

boost_if_changed(int changed, int mom, int dad, int opcode)

The **crossover** procedure calls **boost_if_changed(int changed, int mom, int dad, int opcode)** if the crossover operation has created offspring plans that are different from their parents. In turn, **boost_if_changed(int changed, int mom, int dad, int opcode)** calls **boost_oprob(PLAN *p, int op)** to affect operator probability changes in plans contributing new offspring.

parallel.c

The **parallel.c** file contains the **int process_in_parallel(int dummy, int i)** function. Procedure **evaluate()** (in **evaluate.c**) calls this function (via Uniform System call **GenOnI**) when the user compiles SAMUEL for the Butterfly's Mach Uniform System. This function permits population members to be evaluated in parallel.

reset.c

The **reset.c** file contains the **reset_ga()** procedure. This procedure updates GA variables between experiments.

select.c

The **select.c** file contains the **select()** procedure. The **generate()** procedure calls **select()** after calling **evaluate()** and **measure()**. The **select()** procedure first computes the fitness of each plan in the population. When maximizing fitness (i.e., **Maxflag** is set via the **max** flag in the **params** file), a plan's fitness is non-zero if the plan's performance is greater than a baseline performance (i.e., greater than **Baseline**). In this case, the fitness is the plan's performance value minus **Baseline**. When minimizing fitness, a plan's fitness is non-zero if the plan's performance is less than **Baseline**; thus, fitness is **Baseline** minus the plan's performance.

Next, **select()** computes the average fitness. Based on this average, **select()** determines each plan's expected number of offspring. A plan's expected number of offspring is a plan's fitness divided by the average fitness. When all population members have a fitness value of zero, each plan's fitness is set to one along with the average population fitness.

Using a distribution based on each plan's expected number of offspring, the function randomly selects the number of offspring each plan contributes to the new generation. This is called "roulette wheel selection" in the GA literature (Baker, 1987). The "parent1" field of each new population member holds the index of of the parent plan. To increase the likelihood of creating new plans when **generate()** calls **crossover()** later on, **select()** randomly shuffles these parent indices. Finally, by using the parent indices as pointers, **select()** copies the contents of old plans (**Old[plan index]** to new plans (**New[plan index]** by calling **copy_plan(PLAN *p, PLAN *q)**.

specialize.c

The **specialize.c** file contains a procedure and a related function: **specialize(int i)** and **int specialize_plan(RULE *rule, int length)**. The overall structure of the file and its routines is similar to the structure of **generalize.c**.

The **SPECLIST[plan index]** (the same as the SPEC vector of the operator adjustment list **Olist[MAX_OPERATORS][MAX_POPSIZE]**) holds the list of all plans requiring the application of the *specialize* operator. The **SPECMAX** (the same as the SPEC dimension of **Onext[MAX_OPERATORS]**) holds a count of the number of offspring requiring the application of the operator.

Procedure **evaluate()** calls **specialize(int i)** within an iterative loop over the population of plans, passing the plan's index each time. This differs from the **crossover()** or **mutate()**

procedures, where the iterative loop over plans is in the operator procedure itself. As a result, **specialize(int i)** first needs to find the plan's comparable **SPECLIST** index given the plan's population index. The procedure saves the current number of rules in the plan (in **oldlength**) before calling function **specialize_plan(RULE *rule, int length)** to perform the actual *specialize* operation. This function returns the resulting number of rules in the plan and **specialize(int i)** stores this number in **newlength**. At the bottom of the loop, if there is a change in the plan's length, **specialize(int i)** stores the percent usage of the **specialize** operator in the population in the **op_pcnt[MAX_OPERATORS]** array, and then saves **newlength** in the plan's length field. If the **Boost_op_trace** flag is set (via setting the **boost_trace** parameter in the **params** file), a trace statement reports whether or not rules were specialized in the current plan.

Function **int specialize_plan(RULE *rule, int length)** examines the episodes associated with a plan. Starting with the first episode as a basis of comparison, the function performs a linear search for an episode that has a higher payoff at least as high as **Good_payoff (set via the the good_payoff parameter in params). If the search is successful, the function attempts to specialize rules in the episode. This episode then becomes the new basis of comparison in finding an episode having an even higher payoff.**

When **int specialize(RULE *rule, int length)** finds a better performing episode, the function tests each rule in the episode to see if it is a maximally general rule (by calling **general_rule(RULE *rule)**). If so, the function copies the fired rule into a new rule structure, and then specializes each condition atom to cover the sensor value experienced when the rule fired (recorded in the **History.sensors[decision step][condition index]**) by calling **specialize_atom(*attr, ATTRIBUTE *atom, unsigned int value, ATTRIBUTE oldatom)**.

After specializing a rule, **int specialize(RULE *rule, int length)** initializes the specialized rule, indicating the rule's parent, creation time, and other associated properties. If the new rule is unique, then it is added to the plan.

APPENDIX D5: EM FILES (WORLD MODEL FILES)

em.c

The **em.c** file contains the procedures and functions that simulate the EM world model, and utility functions used in updating the visual display. Procedures essential to the CPS's interface with the world model are **init_env(unsigned int seed), read_sensors(), set_action(int action, int op), take_action(), double get_reward(), int end_of_episode(),** and **reset_env()**. See Chapter 8 for a discussion on how to implement a world model using these procedures and functions.

The local variables used in modeling EM are placed at the top of the **em.c** file. There is no separate **em.h** file. File **em.c** also defines the array **PARAM_TABLE env_params[]**, which sets up the correspondence between input parameters and EM variables. This correspondence is also documented in the **params.def** file.

A few data structures from **cps.h** are declared external at the top of **em.c**. These data structures are not essential to the EM world model; they are used for display output and output to the log file **Detailfile**.

init_env(unsigned int seed)

The **init_env(unsigned int seed)** procedure sets up the environmental variables for a fresh run. In particular, the procedure initializes the pseudo-random number generator seeds **em_seed** and **noise_seed**. Procedure **init_env(unsigned int seed)** also initializes EM internal variables relating to the payoff of an episode: **reward, avereward,** and ***totalreward***. The procedure also initializes state variables such as **winrate** and **successes**.

read_sensors()

The **read_sensors()** procedure first updates the world state in terms of the sensor readings and then records these readings into the **Sensor[i]** array, where **i** is the sensor index (by calling **record_sensor(int i, int ptr_type, void *ptr)**). In the EM world model, **read_sensors()** updates the graphical display if **Drawflag** is set. If the **Detail** is greater than zero, the function writes out sensor values at one of two levels of detail to **Detailfile**. (In general, three levels of detail are available. Levels 1 and 2 are combined in procedure **read_sensors()**.)

set_action(int action, int op)

The **set_action(int action, int op)** procedure transforms an action-value of a rule in **cps()** to a form that can be used by the **take_action()** procedure in **em.c**. The **cps()** procedure passes the action-value of a control and the control's index to **set_action(int action, int op)** through the **action** and **op** arguments, respectively. The **set_action()** procedure has several options for defining the EM variable **turn**, depending on the user interface specified through the **Control** variable. (The user specifies the **Control** variable using **control** parameter in the **params** file. See Appendix A3.) If **Control** specifies the RULES or PAUSE interfaces, **set_action(int action, int op)** maps the **action** argument having the range [0 , 9] into the EM variable **turn** having the range [-4, 4]. The **PAUSE** interface permits the user to suspend the simulation. The **RAND_MOVE** interface randomly selects a turn value over the range [-4, 4]. The **USER** interface permits a variety of responses depending upon a user's keyboard input (including random selection). The default response is to decrease **turn** by 4 until the user enters a command.

take_action()

In the EM domain, the **take_action()** procedure contains the dynamical equations governing the movements of the plane and missile. The **take_action()** procedure models the effect of the turn over a number of time steps maintained in the **substeps** counter. The simulation of the turn ends when (1) the missile hits the plane (**endflag** is set), (2) **substeps** exceeds the maximum time limit defined in **Tstep**, or (3) the missile's speed (in **mspeed**) falls below the minimum speed limit (in **Mspeed_min**). The missile hits the plane if the distance between the missile and plane is less than **Safe_radius**. If **Drawflag** is a value from 1 to 4, procedure **take_action()** also displays the missile and plane by calling **draw_plane()** and **draw_missile()**, respectively. The view depends on the **Drawflag** option selected. A user sets variable **Drawflag** by setting the **draw** parameter in the **params** file. (See Appendix A3.)

double get_reward()

Function **double get_reward()** returns **reward** at the end of an episode to calling procedure **cps()**. Procedure **cps()** assigns the returned value to the CPS variable **Reward**. The reward is 1000 if the plane is not hit; otherwise, the reward is (10)**step** + **substep**. The function also maintains the **successes** and **totalreward** counters, and EM's **episodes** counter. Function **double get_reward()** also calculates the average reward per episode (**avereward**) and the percentage of successes per episode (**winrate**). If the **Drawflag** is set, the function updates the display; if the **Detail** is greater than zero, the function writes a summary of the episode at one of three levels of detail to **Detailfile**.

int end_of_episode()

Function **int end_of_episode()** returns the world model's **endflag** to the CPS each step of an episode.

update_environment()

Generally, **update_environment()** implements any background environmental changes occurring in the environment. In EM world model, the **update_environment()** simply increments the episode's decision **step** counter (corresponding to the CPS's **Step** variable).

reset_env()

The **reset_env()** function performs several "reset" operations: (1) clears state variables **endflag**, **hit**, and **substep**, (2) reinitializes display variables, and (3) reinitializes states variables for the plane and missile. Certain plane and missile state variables are reset based on the range limits specified in the **params** file. For example, **reset_env()** randomly selects an initial value for **bearing** over the range **Bear_lo** to **Bear_hi**. After initialization, if **Drawflag** is set,

reset_env() also calls **draw_plane()** and **draw_:nissile()**.

history.c

The **history.c** file contains the **print_history(FILE *fp, HISTORY *h)** and the **read_history(FILE *fp, HISTORY *h)** procedures. The **dump_env(char *filename)** and **restore_env(char *filename)** procedures in **em.c** call the **print_history(FILE *fp, HISTORY *h)** and the **read_history(FILE *fp, HISTORY *h)** procedures, respectively. These lines of code are currently not being used (and are therefore commented out).

APPENDIX D6: UTILITY FILES AND THEIR PROCEDURES

emfont

The **emfont** file contains a special font used in the EM world model display. A user can select the font by setting the **draw** parameter in the **params** file to 4. (See Appendix A3.)

atoms.c

The **atoms.c** file provides several utilities that the CPS uses for processing atoms.

read_atom(ATTRIBUTE *attribute, ATOM *atom, char *s)

Procedure **read_atom(ATTRIBUTE *attribute, ATOM *atom, char *s)** reads each atom of a rule during the execution of a while loop in **read_rule FILE *fp, RULE *r)** (in **rules.c**). The first argument to **read_atom (ATTRIBUTE *attribute, ATOM *atom, char *s)** specifies the attribute (sensor or control); the second argument points to the condition values, and the last argument specifies the name of the atom.

print_atom(FILE *fp, ATTRIBUTE *attribute, ATOM atom)

Procedure print_rule(FILE *fp, RULE *r) (in **rules.c**) calls **print_atom(FILE *fp, ATTRIBUTE *attribute, ATOM atom)** for each condition and action atom of a rule.

match_atom(ATTRIBUTE *attribute, ATOM atom, int value)

Procedure **match_atom(ATTRIBUTE *attribute, ATOM atom, int value)** matches the current sensor value against the lower and upper limits specified in an atom. The first argument points to the condition atom; the second argument points to the condition values, and the last argument specifies the current value of a sensor. Procedures **find_matches()** (in **cps.c**) calls **match_atom(ATTRIBUTE *attribute, ATOM atom, int value)** to generate a match set. Subsequently, **resolve_conflicts(int action)** calls the matching procedure while determining the winning rule in the match set. Procedure **subsumed(RULE *r, RULE *q)** (in **rules.c**) also calls **match_atom(ATTRIBUTE *attribute, ATOM atom, int value)** while testing to see if atoms in rules having a **PATTERN** type of attribute subsume one another. The **int generalize_plan(RULE *rule, int length)** function calls **match_atom(ATTRIBUTE *attribute, ATOM atom, int value)** during rule generalization.

mutate_atom(ATTRIBUTE *attribute, ATOM *atom)

Function **char mutate_plan(PLAN *p)** (in **mutate.c**) calls **mutate_atom(ATTRIBUTE *attribute, ATOM *atom)** to mutate the condition and action atoms of rules in plans. The first argument of **mutate_atom(ATTRIBUTE *attribute, ATOM *atom)** points to the specific sensor or control atom; the second arguments points to the condition or action values. Procedure **mutate_atom(ATTRIBUTE *attribute, ATOM *atom)** mutates an atom depending on its type: **LINEAR, CYCLIC, STRUCTURED,** or **PATTERN**. In all cases, the function randomly selects whether to mutate the lower or upper bound of the mutate atom. If **Single_act** is set, **mutate_atom(ATTRIBUTE *attribute, ATOM *atom)** mutates the upper bound and then sets the lower bound to the same value.

creep_atom(ATTRIBUTE *attribute, ATOM *atom)

Procedure **char creep_plan(PLAN *p)** (in **creep.c**) calls **creep_atom(ATTRIBUTE *attribute, ATOM *atom)** to apply creep to both the condition and action atoms of rules in plans. This procedure is similar to **mutate_atom(ATTRIBUTE *attribute, ATOM *atom)**, except that bounds are modified by incrementing or decrementing the existing values rather than by substituting new randomly selected values.

make_general_atom(ATTRIBUTE *attribute, ATOM *atom)

Procedure **make_general_atom(ATTRIBUTE *attribute, ATOM *atom)** creates a general atom having bounds that cover the range of the attribute. For **LINEAR** and **CYCLIC** atoms, the value of the lower bound is zero, and the value of the upper bound is the number of attributes values minus 1. For **STRUCTURED** and **PATTERN** atoms, both the upper and lower bounds are zero. The **reset_ga()** procedure calls **make_general_atom(ATTRIBUTE *attribute, ATOM *atom)** to make a generally maximal rule during program initialization if **Initflag** is less than two. The **read_rule(FILE *fp, RULE *r)** procedure also calls **make_general_atom(ATTRIBUTE *attribute, ATOM *atom)** for all conditions and actions of a rule before reading in condition and action boundary values for the atoms of the rule.

specialize_atom(ATTRIBUTE *attr, ATOM *atom, int value, ATOM oldatom)

Function **int specialize_plan(RULE *rule, int length)** (in **specialize.c**) calls **specialize_atom(ATTRIBUTE *attr, ATOM *atom, int value, ATOM oldatom)** to specialize each condition of the rule, i.e., define the bounds for each rule condition, given the current sensor values. The first argument points to the condition atom being specialized; the second argument points to the condition values, the third argument specifies the current value of the corresponding sensor, and the fourth argument is the original atom before applying the *specialize* operator. Procedure **specialize_atom(ATTRIBUTE *attribute, ATOM *atom)** specializes an atom depending on its type: **LINEAR**, **CYCLIC**, **STRUCTURED**, or **PATTERN**. In all cases, the procedure determines the upper and lower bounds of the specialized atom by covering the current sensor reading with upper and lower bounds that significantly reduce the range covered by the "old atom."

print_atom_value(ATTRIBUTE *attr, unsigned int value)

The **print_atom_value(ATTRIBUTE *attr, unsigned int value)** procedure prints out a sensor value associated with an atom. If **Traceflag** is set, procedures **int specialize_plan(RULE *rule, int length)** and **int generalize_plan(RULE *rule, int length)** call **print_atom_value(ATTRIBUTE *attr, unsigned int value)**.

generalize_atom(ATTRIBUTE *attr, ATOM *atom, int value)

Function **int generalize_plan(RULE *rule, int length)** (in **generalize.c**) calls **generalize_atom(ATTRIBUTE *attr, ATOM *atom, int value, ATOM oldatom)** to generalize each condition of the rule just enough to cover the current sensor values. The first argument points to the condition atom being generalized; the second argument points to the condition values, and the third argument specifies the current value of the corresponding sensor. Procedure **generalize_atom(ATTRIBUTE *attribute, ATOM *atom)** generalizes an atom depending on its type: **LINEAR, CYCLIC, STRUCTURED,** or **PATTERN.** In all cases, the procedure expands the upper and lower bounds of the atom enough to cover the current sensor reading.

best.c

The **best.c** file contain utilities used by the GA for saving the best current plan into the dynamic array **Bestset** and printing out the **Bestset** of plans. The file includes the procedures **savebest(), savebest_plan(PLAN *p),** and **printbest().** Every **Best_interval** generations, **savebest()** clears array **Bestset** to reinitialize it with the current best plan in the population. The **savebest()** function calls **savebest_plan()** to save the current best plan. Function **int eval(PLAN *p)** (in file **eval.c**) also calls **savebest_plan()** after CPS evaluates the plan. Every **Graph_rate** generations, **generate()** (the main driver for the GA) calls **printbest()** to write out an output file of the current best plans in the **Bestset.** These high performing plans are evaluated later on using longer episodes.

files.c

The **files.c** file contains functions for opening and closing streams which may or may not be the standard input/output files.

memory.c

The **memory.c** file contains versions of **malloc** for both standard UNIX and the Mach Uniform System on the Butterfly.

params.c

The **params.c** file contains utilities for processing the input parameters in the **params** file. These procedures include **get_params(PARAM_TABLE *pt), find_param(PARAM_TABLE *pt, char *s),** and **set_param(FILE *fp, PARAM_TABLE *pt, int n).**

plan.c

The **plan.c** file contains utilities the GA uses for printing, reading, and copying plans. These functions include **print_plan(FILE *fp, PLAN *p), read_plan(FILE *fp, PLAN *p), print_compact_plan(FILE *fp, PLAN *p), read_compact_plan(FILE *fp, PLAN *p),** and **copy_plan(PLAN *p, PLAN *q).** In the "compact" plan representation, the GA only stores numerical information; e.g., the file does not contain the names of the sensors and actions, nor the names of each rule's associated properties. The compact representation is used in storing plans to "best" files, since these files store a tremendous amount of data. The non-compact read and print functions represent plans as if-then rules that are easily read by a user.

rules.c

The **rules.c** file contains utilities the GA uses for processing rules, including **compile_rules()** for organizing rules into a structure the CPS can use, **copy_rule(**FILE ***fp,** RULE ***r), print_rule(**FILE ***fp,** RULE ***r),** and **read_rule(**FILE ***fp,** RULE ***r).**

save.c

The **save.c** file contains utilities that saves the current state of the GA, including a genealogical trace of the plans at that time. Procedure **save_state()** calls **save(char *filename)** periodically if the number of save files, **Nsave,** is greater than zero and the **Save_interval** is specified. (The **Save_interval** and the **Nsaves** variables are initialized through the **save_interval** and **save** parameters in the **params** file, respectively.) The **save(char *filename)** procedure is also called at the end of the run if **Lastflag** is set. (**Lastflag** is set through the **lastflag** parameter in the **params** file.) The **save(char *filename)** writes out the file to **Savefile**. (**Savefile** is specified through the **savefile** parameter in the **params** file.) Finally, **save_state()** calls **geneology()**. If the **Geneology** flag is set, **geneology** writes out the current generation's plans in a compact format to file **geneology**. (The **Geneology** flag is specified through parameter **geneology** in the **params** file.) The user should exercise care in setting the **Geneology** flag; **geneology()** generates a tremendous amount of output.

sets.c

The **sets.c** file contains the following domain-independent set manipulation procedures: **init_set_length(int n)** obtains the set length in terms of integers given bit length n; **set_to_empty(register SET s)**, sets elements of the set s to zero; **set_insert(register int n, register SET s)**, inserts integer **n** into set s; **set_is_null(register SET s)**, tests whether or not set s is null; **set_intersect(register SET s1, register SET s2)**, "ANDs" set **s1** with set s2 and places the result in set **s1**; **set_union(register SET s1, register SET s2)**, "ORs" set s1 with set s2 and places the result in set **s1**; **set_assign(register SET s1, register SET s2)**, writes set s2 to set s1; **set_delete(register int n, register** SET s), deletes, if possible, **n** from set s; **set_extract(register int *list, register** SET s), removes a list of elements from set s, and **set_to_full(register** SET s), pads set s with zeros.

APPENDIX D7: SHELL SCRIPT UTILITIES

Appendix D6 lists the shell scripts available and the purpose of each. See Chapter 7 for more information on how to use these scripts.

ch

Script **ch** permits a user to update a line in the **params** file.

demo

Script **demo** file contains a script for running the SAMUEL demo.

avegraph

The **avegraph** file generates the **graph.all** and **graph.ave** files. The **graph.all** file summarizes the results over all of the experimental runs; **graph.ave** file is the experimental average of **graph.all**. The **avegraph** script is used in **run-samuel**.

getindex

The **getindex** file contains a script that filters out the index of the best plan from each **graph** file generated in the **run-samuel** script.

mkgraph

The **mkgraph** file contains a script filters out the best plan for each epoch, given the extended evaluations of several plans each epoch (extended evaluations from the **retest** program). The **mkgraph** script is used in **run-samuel**.

p

The **p** script executes the UNIX **more** command taking **params** as an argument.

run-samuel

Script **run-samuel** permits a user to run a set of experiments in both "training" and "testing" mode.

ttest

Script **ttest** script performs a student t-test between the data points of two summary **graph** files generated by **run-samuel**.

show

The **show** script permits a user view one or two **graph** files on a Sun workstation using the **eview** command. This command uses the **plot** file.

wins

The **wins** file contains an awk script summarizing the **trace** file.

APPENDIX D8: PROGRAM INPUT FILES

attributes

The **attributes** file lists the attributes of each sensor and control. See Appendix B for an explanation of the **attributes** file.

init

The **init** file specifies the rules to be placed in the initial plan of SAMUEL if the **init** parameter is 1 or 2.

params

The **params** file contains all of the runtime parameters for SAMUEL. See Appendix A for a complete listing of these parameters.

rules

The **rules** file contains rules for the initial population when running SAMUEL without the GA.

REFERENCES

Baker, 1987. Baker, J. E., "Reducing bias and inefficiency in the selection algorithm," *Genetic Algorithms and Their Applications: Proceedings of the Second Inter- national Conference on Genetic Algorithms*, 14-21 (1987).

De Jong, 1975. De Jong, Kenneth A., "An analysis of the behavior of a class of genetic adaptive systems," Doctoral Dissertation, University of Michigan, Ann Arbor, Michigan (1975).

Grefenstette, 1983. Grefenstette, John J. "A user's guide to GENESIS," *Technical Report CS-83-11, Computer Science Department*, Vanderbuilt University (1983).

Grefenstette, 1986. Grefenstette, John J. "Optimization of control parameters for genetic algorithms," *IEEE Trans. Systems, Man, and Cybernetics*, **SMC-16(1)** , pp. 122-128 (1986).

Grefenstette, 1988. Grefenstette, John J. "Credit assignment in rule discovery system based on genetic algorithm," *Machine Learning* **3(2/3)**

Grefenstette, 1989. Grefenstette, John J., "A system for learning control strategies with genetic algorithms," *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications, June 4 - 7, 1989 at the George Mason University, Fairfax, VA. ed. J. David Schaffer, Morgan Kaufmann, San Mateo, California (1989)*

Grefenstette, 1990. Grefenstette, John J., Ramsey, Connie Loggia, and Schultz, Alan C., "Learning Sequential Decision Rules Using Simuation Models and Competition," Machine Learning Journal, (1990).

Holland, 1975. Holland, John H., *Adaptation in natural and artificial systems," University of Michigan Press, Ann Arbor, Michigan (1975).*

Holland, 1986. Holland, John H., Holyoak, Keith J., Nisbett, Richard E., Thagard, Paul R., *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, Cambridge, Massachusetts (1986).

Riolo, 1988. Riolo, Rick L., *Empirical Studies of Default Hierarchies and Sequences of Rules in Learning Classifier Systems*, Doctoral Dissertation, University of Michigan (1988).

Schultz, 1990. Schultz, Alan and Grefenstette, John J. "Improving Tactical Plans with Genetic Algorithms", *Second International Conference on Tools for Artificial Intelligence*, IEEE, (1990).

Wilson, 1987. Wilson, S. W., "Hierarchical credit allocation in a classifier system," *Genetic Algorithms and Simulated Annealing*, ed. L. Davis, Pitman, London, UK (1987).